The Thesis Committee for Kaiyuan Wang
certifies that this is the approved version of the following thesis:

MuAlloy: An Automated Mutation System for Alloy

APPROVED BY

SUPERVISING COMMITTEE:

Sarfraz Khurshid, Supervisor

Dewayne Perry

MuAlloy: An Automated Mutation System for Alloy

by

Kaiyuan Wang, B.E.

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN ENGINEERING

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2015

Dedicated to my loving and supportive parents, Zhen Wang and Rong Xie.

Acknowledgments

I wish to thank my supervisor Professor Sarfraz Khurshid for his genius guidance and supportive encouragement throughout my master thesis. I would also like to thank Professor Dewayne Perry for taking the time to read and evaluate my work.

Besides my thesis committee members, I would like to thank Kewei Ma. Kewei is my teammate in a course project in the Verification and Validation of Software class. The idea of this thesis is inspired in part based on that previous work on Alloy.

This work was funded in part by the National Science Foundation (NSF Grant Nos. CCF-0845628 and CNS-1239498).

MuAlloy: An Automated Mutation System for Alloy

Kaiyuan Wang, M.S.E
The University of Texas at Austin, 2015

Supervisor: Sarfraz Khurshid

Mutation is a powerful technique that researchers have studied for several decades in the context of imperative code. For example, mutation testing is commonly considered a "gold standard" for test suite quality. Mutation in the context of declarative languages is a less studied problem. This thesis introduces a foundation for mutation-driven analyses for Alloy, a first-order, declarative language based on relations. Specifically, we introduce a family of mutation operators for Alloy models and define algorithms for applying the operators on different parts of the models. We embody these operators and algorithms in our prototype tool μAlloy that provides a GUI-based front-end for customizing the application of mutation operators. To demonstrate the potential of our approach, we illustrate the use of μAlloy in two application scenarios: (1) mutation testing for Alloy (in the spirit of traditional mutation testing for imperative languages); and (2) program repair for Alloy using mutation.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Mutation, i.e., syntactic modification, is a well-studied technique for testing and debugging imperative code [8, 9, 20]. For example, *mutation testing* provides a powerful method for evaluating quality of test suites as well as for generating high quality suites [9, 20]. While much of the original work on mutation focused on testing, more recent work has shown how to utilize mutation for *debugging*, specifically for repair of faulty imperative code [8].

In contrast to the plethora of work on mutation for imperative code, mutation in the context of declarative languages is a less studied problem area with existing techniques focusing largely on mutating declarative specifications of imperative programs with the purpose of testing these imperative programs more thoroughly [3, 4]. A key reason for this lack of prior work on mutation for the specific purpose of analyzing declarative code is that traditional techniques for testing declarative code are quite different (both conceptually and in practice) from techniques for testing imperative code. For example, notions as fundamental as test case, test execution, and test pass/fail result, which occur naturally for imperative code, do not have obvious definitions for declarative code [24].

Our thesis is that mutation has an important role in testing and debugging of declarative code. We introduce a foundation for mutation-driven analyses for Alloy, a first-order, declarative language based on relations. Specifically, we introduce a family of mutation operators for Alloy models and define algorithms for applying the operators on different parts of the models. The operators address different constructs, e.g., *signature* declarations, formulas, and commands, of the Alloy language. The algorithms guarantee generation of valid Alloy models as mutants.

μAlloy embodies our mutation operators and algorithms, and provides a GUI-based front-end for customizing the application of mutation operators. To demonstrate the potential of our approach, we illustrate the use of μAlloy in three application scenarios: (1) mutation testing for Alloy (in the spirit of traditional mutation testing for imperative languages); and (2) program repair for Alloy using mutation.

We make the following contributions:

- **Mutation for Alloy.** We introduce the idea of using mutation as a basic technique for analysis in the context of the declarative language Alloy;
- **Mutation operators for Alloy.** We introduce a family of mutation operators based on the Alloy language syntax and semantics;
- **Algorithms for generating mutant Alloy models.** We present algorithms to create mutants of given Alloy models

- **μAlloy tool.** We present our tool μAlloy, which embodies the algorithms and allows the user to customize mutation;
- **Applications.** We illustrate two analyses enabled by μAlloy: (1) mutation testing for Alloy (in the spirit of traditional mutation testing for imperative languages); and (2) program repair for Alloy using mutation.

Chapter 2

Background

Alloy is a first-order, declarative language based on relations [1]. The Alloy tool-set provides a SAT-based back-end analyzer for Alloy models. Given an Alloy model, the analyzer translates the model to a propositional formula that is solved using an off-the-shelf SAT solver; the SAT solution is translated to an Alloy *instance*, which represents a satisfying assignment to the sets and relations in the model with respect to its formulas. The user may choose to enumerate multiple instances, say to gain more confidence in the correctness of the model. The Alloy analyzer performs its analysis for a given *scope*, i.e., bound on the universe of discourse. The user may provide a desired scope by specifying an upper bound on the number of atoms for each basic set or use the default scope of 3.

An Alloy model consists of three basic parts:

- *Set and relation declarations.*

  A *signature* declaration in Alloy introduces a set of atoms and may introduce zero or more relations with arity at least 2, as well as cardinality constraints on the relations. A signature declaration optionally defines a fact which restricts its relations. For example, the Alloy code "***sig*** *List*

{ *size:* ***set*** *Int* }{ ***one*** *size* }" declares a set of *List* atoms and a binary relation "*size: List* $\times$ *Int*" where each *List* only has one *size*.

- *Formula paragraphs.*

  A formula paragraph introduces a constraint on the sets and relations declared in the model. A *fact* paragraph declares a set of constraints that must always hold. A *pred* paragraph is a named and optionally parameterized constraint that can be invoked in different contexts. A *fun* paragraph is a named and optionally parameterized expression that can be invoked in different contexts; a *fun* declaration has a return type corresponding to the expression. An *assert* paragraph is a constraint that is expected to follow from the facts of a model; assertions are typically used to check properties of Alloy models to validate them.

- *Commands*

  A command instructs the Alloy analyzer what constraints to solve using its SAT-based back-end. A *run* command states a predicate or a function that the analyzer *simulates*, i.e., the constraint solving problem is to find a solution to the constraint represented by the body of the predicate (or function) and any additional constraints enforced by the facts. A *check* command causes the analyzer to search for a counterexample showing that an assertion does not hold. Each command (implicitly or explicitly) specifies a *scope*, and the instances or counterexamples generated are within that scope. Moreover, each command may specify an

expected outcome in terms of constraint satisfiability using the "expect k" clause where $k = 0$ states the analyzer is expected to find no instance or counterexample and $k \geq 1$ states the analyzer is expected to find at least one instance or counterexample (k does not specify the number of solutions).

Given an Alloy model, the analyzer executes the selected command(s) by translating the corresponding Alloy formulas into propositional formulas, which are solved using SAT. If an instance or a counterexample is found, the user can inspect it in textual format, in graphical format or as a instance tree. The user may also iterate through the solutions and enhance her/his confidence in the correctness of the model. To improve efficiency, Alloy analyzer provides symmetry-breaking where isomorphic solutions are removed so that the total number of solutions is reduced [22].

Next, we present two Alloy models to introduce some key Alloy constructs that muAlloy handles.

Figure 2.1 presents a small Alloy model for singly-linked, acyclic lists; specifically, the model checks if predicate *RepOk* and predicate *RepOk2* are equivalent. The keyword **module** names the model as *linkedlist*. The signature declarations in lines 3–5 and 7–12 introduce *List* and *Node* as sets of atoms, and *header* and *link* as binary relations that have type $List \times Node$ and type $Node \times Node$, respectively. Signature **sig** *List* defines a relation *header* and **sig** *Node* defines a relation *link*. The relation *header* has a multiplicity constraint

```
1  module linkedlist
2
3  sig List {
4    header: lone Node
5  }
6
7  sig Node {
8    link: set Node
9  }
10 {
11   lone link
12 }
13
14 pred RepOk(l: List) {
15   all n: l.header.*link | n !in n.^link
16 }
17
18 pred RepOk2(l: List) {
19   no l.header || some n: l.header.*link | no n.link
20 }
21
22 assert Equiv {
23   all l: List | RepOk[l] <=> RepOk2[l]
24 }
25
26 check Equiv for 3 but 1 List
```

Figure 2.1: Alloy model for singly linked list.

lone so that each *List* atom has zero or one header. The relation *link* also has a multiplicity constraint **set** along with a signature fact declared in line 11, which together restrict that each *Node* relates to zero or one *Node* through *link*. The predicate *RepOk* in lines 14–16 uses universal quantification to define acyclicity. The body of the universal quantification states that for any node n reachable from the *header* of a list (including the header itself), n is excluded from the set of all nodes reachable from n following one or more traversals

```
1  module binary_tree
2
3  one sig BinaryTree {
4    root: lone Node
5  }
6
7  sig Node {
8    left: lone Node,
9    right: lone Node
10 }
11
12 fact {
13   BinaryTree.root.*(left + right) = Node
14 }
15
16 pred RepOk(t: BinaryTree) {
17   all n: t.root.*(left + right) {
18     n !in n.^(left + right)
19     no n.left & n.right
20     lone n.~(left + right)
21   }
22 }
23
24 run RepOk for exactly 1 BinaryTree, 5 Node expect 1
```

Figure 2.2: Alloy model for binary tree.

along *link*. Thus, *RepOk* specifies that no list has a cycle. The predicate *RepOk2* in lines 18–20 states that for *List l*, it either has no header or there exists a *Node n* reachable from *l*'s header where no other *Node* appears after *n* through *link*. The assertion in lines 22–24 checks if predicates *RepOk* and *RepOk2* are equivalent for the given scope. The *check* command in line 26 instructs the analyzer to find a counterexample for the assertion *Equiv* using a scope of up to 3 *Node* and up to 1 *List*.

Figure 2.2 presents another Alloy model, which represents binary trees.

Specifically, the model generates binary tree instances for users within the given scope. The module declaration in line 1 assigns the model with a name *binary_tree*. The signature declarations in lines 3–5 and lines 7–10 introduce a singleton set *BinaryTree* and a set of *Node* atoms. A *BinaryTree* has zero or one root of type *Node*. A *Node* has zero or one left child and zero or one right child. The fact in lines 12–14 states that all nodes reachable from the root of the singleton atom *BinaryTree* are equal to the set of all *Node* atoms. The predicate *RepOk* in lines 16–22 restricts the model as follows: (1) be acyclic; (2) given any *Node*, its left child and right child do not point to the same node; and (3) any node in the binary tree does not have more than one parent node. The command **run** in line 24 invokes predicate *RepOk* and sets a bound of the model with exactly 1 *BinaryTree* and up to 5 *Node*. The command also tells the Alloy Analyzer that at least one instance should be found.

Chapter 3

Example mutants of Alloy models

This section shows some example mutants for the two Alloy models introduced in Chapter 2 (Figures 2.1 and 2.2).

A mutant for an Alloy model is created by applying exactly one mutation operator on the model. A *valid* mutant is a mutant that does not yield syntax errors, type errors, compilation warnings or any type of errors that prevents Alloy analyzer from performing its analysis.

Figure 3.1 shows a valid mutant for the singly linked list model. It is generated by replacing signature multiplicity in line 3 from "***(set) sig*** *List*" to "***one sig*** *List*".

Figure 3.2 presents another valid mutant for the singly linked list model where the cardinality keyword "**lone**" in line 4 is replaced with "**one**".

Figure 3.3 shows a valid mutant of singly linked list model which is generated by replacing reflexive transitive closure "*" in line 15 with transitive closure "^".

Figure 3.4 is a valid mutant for the singly linked list model where the binary operator bi-implication "<=>" in line 23 is replaced with implication "=>".

```
1  module linkedlist
2
3  one sig List { // from ``set'' to ``one''
4    header: lone Node
5  }
6
7  sig Node {
8    link: set Node
9  }
10 {
11   lone link
12 }
13
14 pred RepOk(l: List) {
15   all n: l.header.*link | n !in n.^link
16 }
17
18 pred RepOk2(l: List) {
19   no l.header || some n: l.header.*link | no n.link
20 }
21
22 assert Equiv {
23   all l: List | RepOk[l] <=> RepOk2[l]
24 }
25
26 check Equiv for 3 but 1 List
```

Figure 3.1: Mutating signature multiplicity for singly linked list model.

The valid mutant in Figure 3.5 derives from the binary tree model and it is generated by deleting the "***expect*** *1*" in line 26.

Figure 3.6 presents an invalid mutant for the binary tree model. The model is created by replacing the quantifier declaration "***all*** *n: t.root.*(left + right)*" in line 17 with "***all*** *n:* ***some*** *t.root.*(left + right)*". The Alloy analyzer cannot perform analysis on the mutant since it requires higher-order quantification that could not be skolemized.

```
1  module linkedlist
2
3  sig List {
4    header: one Node // from ``lone'' to ``one''
5  }
6
7  sig Node {
8    link: set Node
9  }
10 {
11   lone link
12 }
13
14 pred RepOk(l: List) {
15   all n: l.header.*link | n !in n.^link
16 }
17
18 pred RepOk2(l: List) {
19   no l.header || some n: l.header.*link | no n.link
20 }
21
22 assert Equiv {
23   all l: List | RepOk[l] <=> RepOk2[l]
24 }
25
26 check Equiv for 3 but 1 List
```

Figure 3.2: Mutating relation for singly linked list model.

```
1  module linkedlist
2
3  sig List {
4    header: lone Node
5  }
6
7  sig Node {
8    link: set Node
9  }
10 {
11   lone link
12 }
13
14 pred RepOk(l: List) {
15   all n: l.header.^link | n !in n.^link // from ``*'' to ``^''
16 }
17
18 pred RepOk2(l: List) {
19   no l.header || some n: l.header.*link | no n.link
20 }
21
22 assert Equiv {
23   all l: List | RepOk[l] <=> RepOk2[l]
24 }
25
26 check Equiv for 3 but 1 List
```

Figure 3.3: Mutating unary operator for singly linked list model.

```
1  module linkedlist
2
3  sig List {
4    header: lone Node
5  }
6
7  sig Node {
8    link: set Node
9  }
10 {
11   lone link
12 }
13
14 pred RepOk(l: List) {
15   all n: l.header.*link | n !in n.^link
16 }
17
18 pred RepOk2(l: List) {
19   no l.header || some n: l.header.*link | no n.link
20 }
21
22 assert Equiv {
23   all l: List | RepOk[l] => RepOk2[l] // from ``<=>'' to ``=>''
24 }
25
26 check Equiv for 3 but 1 List
```

Figure 3.4: Mutating binary operator for singly linked list model.

```
1  module binary_tree
2
3  one sig BinaryTree {
4    root: lone Node
5  }
6
7  sig Node {
8    left: lone Node,
9    right: lone Node
10 }
11
12 fact {
13   BinaryTree.root.*(left + right) = Node
14 }
15
16 pred RepOk(t: BinaryTree) {
17   all n: t.root.*(left + right) {
18     n !in n.^(left + right)
19     no n.left & n.right
20     lone n.~(left + right)
21   }
22 }
23
24 run RepOk for exactly 1 BinaryTree, 5 Node // delete ``expect 1''
```

Figure 3.5: Mutating **expect** for binary tree model.

```
1  module binary_tree
2
3  one sig BinaryTree {
4    root: lone Node
5  }
6
7  sig Node {
8    left: lone Node,
9    right: lone Node
10 }
11
12 fact {
13   BinaryTree.root.*(left + right) = Node
14 }
15
16 pred RepOk(t: BinaryTree) {
17   all n: some t.root.*(left + right) { // from ``one'' to ``some''
18     n !in n.^(left + right)
19     no n.left & n.right
20     lone n.~(left + right)
21   }
22 }
23
24 run RepOk for exactly 1 BinaryTree, 5 Node expect 1
```

Figure 3.6: Invalid mutant for binary tree model.

Chapter 4

Mutation for Alloy

This chapter presents our basic μAlloy approach for mutating Alloy models. Figure 4.1 presents the Alloy 4 grammar that provides the basis for defining mutation operators. Next, we introduce a family of mutation operators for Alloy (Section 4.1). Finally, we present our mutant generation algorithm (Section 4.2).

4.1 Mutation Operators Definition

This section introduces the mutation operators that μAlloy supports. These mutation operators are categorized into 7 groups based on their structural similarity.

4.1.1 Small examples

As an illustrative example mutation operator, consider signature multiplicity, which is defined as

$$\text{mult} ::= \textbf{lone} \mid \textbf{some} \mid \textbf{one}$$

A signature declaration, say "***sig*** *Node* { ... }" can be mutated to "***lone sig*** *Node* { ... }", "***some sig*** *Node* { ... }" or "***one sig*** *Node* { ... }".

alloyModule ::= [moduleDecl] import* paragraph*
moduleDecl ::= **module** qualName [[name,$^+$]]
import ::= **open** qualName [[qualName,$^+$]] [**as** name]
paragraph ::= sigDecl | factDecl | predDecl | funDecl | assertDecl | cmdDecl
sigDecl ::= [**abstract**] [mult] **sig** name,$^+$ [sigExt] **{** decl,* **}** [block]
sigExt ::= **extends** qualName | **in** qualName [+ qualName]*
mult ::= **lone** | **some** | **one**
decl ::= [**disj**] name,$^+$ **:** [**disj**] expr
factDecl ::= **fact** [name] block
predDecl ::= **pred** [qualName **.**] name [paraDecls] block
funDecl ::= **fun** [qualName **.**] name [paraDecls] **:** expr **{** expr **}**
paraDecls ::= **(** decl,* **)** | [decl,*]
assertDecl ::= **assert** [name] block
cmdDecl ::= [name **:**] [**run** | **check**] [qualName | block] [scope]
scope ::= **for** number [**but** typescope,$^+$] | **for** typescope,$^+$
typescope ::= [**exactly**] number qualName
expr ::= const | qualName | @name | **this** | unOp expr | expr binOp expr
| expr arrowOp expr | expr **[** expr,* **]** | expr [**!** | **not**] compareOp expr
| expr (**=>** | **implies**) expr **else** expr | **let** letDecl,$^+$ blockOrBar
| quant decl,$^+$ blockOrBar | **{** decl,$^+$ blockOrBar **}** | **(** expr **)** | block
const ::= [**-**] number | **none** | **univ** | **iden**
unOp ::= **!** | **not** | **no** | mult | **set** | **#** | **~** | ***** | **^**
binOp ::= **||** | **or** | **&&** | **and** | **<=>** | **iff** | **=>** | **implies** | **&** | **+** | **-** | **++** | **<:** | **:>** | .
arrowOp ::= [mult | **set**] **->** [mult | **set**]
compareOp ::= **in** | **=** | **<** | **>** | **=<** | **>=**
letDecl ::= name **=** expr
block ::= **{** expr* **}**
blockOrBar ::= block | bar expr
bar ::= **|**
quant ::= **all** | **no** | **sum** | mult
qualName ::= [**this/**] (name **/**)* name

Figure 4.1: Alloy4 grammar [1]

As another example consider mutating an Alloy expression in a formula. A part of the grammar for an Alloy expression is

$$\text{expr} ::= \text{expr}\ [\texttt{!}\ |\ \textbf{not}]\ \textit{comapreOp}\ \text{expr}$$

where *compareOp* could be "**in**", etc. Thus, the constraint "*n* ***!in*** *n.^link*" can be mutated to "*n* ***in*** *n.^link*".

4.1.2 Overview

By designing mutation operators carefully we can reduce the likelihood of generating invalid mutants. For instance, an unary operator in Alloy can only be replaced with a compatible unary operator to preserve the validity of the model. In general, invalid mutants may have syntax errors, semantic errors or result in warnings during execution. Even if mutation operators are designed with care there are still chances that mutants are invalid models. For example, both "**.**" and "`<=>`" are *binOp*, but mutating "*RepOk[l]* `<=>` *RepOk[l]*" to "*RepOk[l]***.***RepOk[l]*" in Figure 2.1 line 23 yields syntax error. Therefore, mutation operators for Alloy models need to be designed with care and mutants must be validated after generation.

Table 4.1: Mutation Operators

From/To	*To/From*
[abstract] [**lone** | **one** | **some**] sig name	[abstract] [**lone** | **one** | **some**] sig name
iden: [**set** | **lone** | **one** | **some**] expr	iden: [**set** | **lone** | **one** | **some**] expr
expr1 [**set** | **lone** | **one** | **some**] `->` [**set** | **lone** | **one** | **some**] expr2	expr1 [**set** | **lone** | **one** | **some**] `->` [**set** | **lone** | **one** | **some**] expr2
[**no** | **lone** | **one** | **some** | **all**] iden: relation | constraints	[**no** | **lone** | **one** | **some** | **all**] iden: relation | constraints
[**no** | **lone** | **one** | **some**] expr	[**no** | **lone** | **one** | **some**] expr
number	**number** + 1
[**expect** 0 | **expect** int (int `>` 0) | ∅]	[**expect** 0 | **expect** int (int `>` 0) | ∅]
expr1 **in** expr2	expr1 **!in** expr2
expr1 `=` expr2	expr1 `!=` expr2
expr1 [`+` | `&` | `-`] expr2	expr1 [`+` | `&` | `-`] expr2
expr1 `-` expr2	expr2 `-` expr1
expr1 `in` expr2	expr2 `in` expr1
expr1 `!in` expr2	expr2 `!in` expr1
expr1.expr2	expr2.expr1
expr1.[*|^]expr2	expr1.~[*|^]expr2
expr1.*expr2	expr1.^expr2
expr1 `<=>` expr2	expr1 `=>` expr2
expr1 `=>` expr2	expr2 `=>` expr1
expr1 `=>` expr2 **else** expr3	expr1 `=>` expr3 **else** expr2
expr1 `<:` expr2	expr1 `:>` expr2
expr1 `++` expr2	expr2 `++` expr1
expr1 `++` expr2	expr1 `+` expr2
expr1 [`<` | `=<` | `=` | `>=` | `>`] expr2	expr1 [`<` | `=<` | `=` | `>=` | `>`] expr2

Table 4.1 defines the different mutation operators μAlloy introduces. In each row, the first column shows candidate constructs that can be mutated to possible constructs in the second column. We group the constructs so as to reduce the likelihood of creating mutants with syntax errors. For example, if "*expr1* `+` *expr2*" is valid in the original model, then mutating "`+`" to "`-`" also produces a valid expression "*expr1* `-` *expr2*". However, some mutation operators may still lead to invalid Alloy mutants. One example is the mutation operator "*expr1*`.`*expr2* to *expr2*`.`*expr1*", where knowing "*expr1*`.`*expr2*" being valid does not imply that "*expr2*`.`*expr1*" is also valid. Another example would be "*expr1*`.`*expr2* to *expr1*`.`$\sim$*expr2*". Suppose *expr1* is a relation with type $A \times B$ and *expr2* is of type $B \times C$, where $B \neq C$. In this case, "*expr1*`.`*expr2*" represents relation $A \times C$, but "*expr1*`.`$\sim$*expr2*" gives compiler warnings because $A \times B$ cannot relation join with $C \times B$.

The sections that follow describe in more detail the different groups of mutation operators.

4.1.3 Constant

Table 4.2 focuses on mutation operators related to integer constants. Similar to mutation operators for integers in imperative languages such as Java, we define increment and decrement mutation operators for integers in Alloy models.

Table 4.2: Mutate Constant Expression

From	*To*	*From*	*To*
number	**number** + 1	**number**	**number** − 1

4.1.4 Multiplicity

Multiplicity key words "**lone**", "**one**", "**some**" and "**set**" restrict the cardinality of Alloy sets and relations. For example, "***one sig*** *BinaryTree* {...}" in Figure 2.2 states that exactly one *BinaryTree* atom exists in all satisfiable solutions. Moreover, "*root:* ***lone*** *Node*" in Figure 2.2 line 4 states that every *BinaryTree* atom has zero or one root.

Table 4.3 shows all mutation operators designed for multiplicity constraints. For instance, the *BinaryTree* signature declaration "***one sig*** *BinaryTree* {...}" in line 3-5 Figure 2.2 can be modified as "***lone sig*** *BinaryTree* {...}".

Table 4.3: Mutate Multiplicity

From	*To*	*From*	*To*
[abstract] sig	[abstract] lone sig	[abstract] lone sig	[abstract] sig
[abstract] sig	[abstract] some sig	[abstract] some sig	[abstract] sig
[abstract] sig	[abstract] one sig	[abstract] one sig	[abstract] sig
[abstract] lone sig	[abstract] some sig	[abstract] some sig	[abstract] lone sig
[abstract] lone sig	[abstract] one sig	[abstract] one sig	[abstract] lone sig
[abstract] one sig	[abstract] some sig	[abstract] some sig	[abstract] one sig
iden: [set] expr	iden: lone expr	iden: lone expr	iden: [set] expr
iden: [set] expr	iden: one expr	iden: one expr	iden: [set] expr
iden: [set] expr	iden: some expr	iden: some expr	iden: [set] expr
iden: lone expr	iden: one expr	iden: one expr	iden: lone expr
iden: lone expr	iden: some expr	iden: some expr	iden: lone expr
iden: one expr	iden: some expr	iden: some expr	iden: one expr

4.1.5 Quantifier

Quantifiers "**no**", "**lone**", "**one**", "**some**" and "**all**" in a formula quantify the sub-expressions they apply to. For instance, the formula in Figure 2.1 line 15 restricts that for each *Node* atom n reachable from the header of the singly linked list, n is not in the set of nodes that appear after n through the relation *link*.

Table 4.4 includes all mutation operators for Alloy quantifiers. Those operators can be applied to the singly linked list model in Figure 2.1. For example, line 15 in Figure 2.1 can be modified to "***no*** *n: l.header.∗link* | *n* ***!in*** *n.ˆlink*". In addition, more mutants can be generated since quantifier keyword "**all**" can be replaced with "**no**", "**lone**", "**one**" or "**some**", thus 3 more mutants can be generated by modifying line 15 in Figure 2.1.

Table 4.4: Mutate Quantifier Expression

From	*To*	*From*	*To*
no iden: rel | cons	lone iden: rel | cons	lone iden: rel | cons	no iden: rel | cons
no iden: rel | cons	one iden: rel | cons	one iden: rel | cons	no iden: rel | cons
no iden: rel | cons	some iden: rel | cons	some iden: rel | cons	no iden: rel | cons
no iden: rel | cons	all iden: rel | cons	all iden: rel | cons	no iden: rel | cons
lone iden: rel | cons	one iden: rel | cons	one iden: rel | cons	lone iden: rel | cons
lone iden: rel | cons	some iden: rel | cons	some iden: rel | cons	lone iden: rel | cons
lone iden: rel | cons	all iden: rel | cons	all iden: rel | cons	lone iden: rel | cons
one iden: rel | cons	some iden: rel | cons	some iden: rel | cons	one iden: rel | cons
one iden: rel | cons	all iden: rel | cons	all iden: rel | cons	one iden: rel | cons
some iden: rel | cons	all iden: rel | cons	all iden: rel | cons	some iden: rel | cons

4.1.6 Unary Expression

Among all unary operators, some restrict the cardinality of Alloy expressions, while others are used to compute over Alloy relations. For example,

line 19 in Figure 2.2 contains an unary operator "**no**" and it states that given any node in a binary tree, its left child and right child may not point to the same node. In addition, line 15 in Figure 2.1 contains a unary operator reflexive transitive closure "*****" which applies to the relation *link*. Conceptually, the expression "***all** n: l.header.*link*" represents the set of all nodes reachable from the header of a list following zero or more traversals along *link*.

Mutation operators related to Alloy unary operators are defined in Table 4.5. Applying the mutation operators to the binary tree model in Figure 2.2 yields several mutants. For example, a mutant could be generated by modifying line 20 from "***lone** n.~(left + right)*" to "***lone** n.(left + right)*". Another example mutant could be created by altering "***lone** n.~(left + right)*" to "***one** n.~(left + right)*".

Table 4.5: Mutate Unary Expression

From	*To*	*From*	*To*
no expr	lone expr	lone expr	no expr
no expr	one expr	one expr	no expr
no expr	some expr	some expr	no expr
lone expr	one expr	one expr	lone expr
lone expr	some expr	some expr	lone expr
one expr	some expr	some expr	one expr
expr1.expr2	expr1.~expr2	expr1.~expr2	expr1.expr2
expr1.expr2	expr1.*expr2	expr1.*expr2	expr1.expr2
expr1.expr2	expr1.^expr2	expr1.^expr2	expr1.expr2
expr1.*expr2	expr1.^expr2	expr1.^expr2	expr1.*expr2
expr1.*expr2	expr1.*expr2	expr1.*expr2	expr1.*expr2
expr1.^expr2	expr1.~^expr2	expr1.~^expr2	expr1.^expr2

4.1.7 Binary Expression

A binary operator combines its left expression and right expression to produce a new expression. For example, binary operator "**!in**" in Figure 2.1 line 15 combines "*n*" and "*n.^link*" into a new expression "*n* ***!in*** *n.^link*". Another example in Figure 2.1 line 23 shows that binary operator "<=>" restricts predicates *RepOk* and *RepOk2* to be equivalent.

Table 4.6 defines mutation operators for binary operations in Alloy grammar. Basically, mutating binary operators involves replacements of the old binary operators with new binary operators, as well as swapping left sub-expression and right sub-expression of binary operators. For example, line 23 in Figure 2.1 contains a binary operator "<=>" and it can be replaced with "=>". Another example is that the expression "*n* ***!in*** *n.^(left + right)*" in Figure 2.2 line 18 can be altered to "*n.^(left + right)* ***!in*** *n*". As before, some of the mutation operators may yield invalid mutants. For instance, Figure 2.1 line 19 contains expression "***no*** *l.header*" and expression "***no*** *n.link*", but modifying "***no*** *l.header*" to "***no*** *header.l*" yields compilation warning while changing "***no*** *n.link*" to "***no*** *link.n*" does not.

4.1.8 Let Expression

A **let** expression allows a variable to be introduced, to highlight an important sub-expression or make an expression or constraint shorter by factoring out a repeated sub-expression. There is no mutation operators related to the **let** expression itself. However, mutation operators may be applied in

Table 4.6: Mutate Binary Expression

From	*To*	*From*	*To*
expr1 [set] -> lone expr2	expr2 lone -> [set] expr1	expr2 lone -> [set] expr1	expr1 [set] -> lone expr2
expr1 [set] -> one expr2	expr2 one -> [set] expr1	expr2 one -> [set] expr1	expr1 [set] -> one expr2
expr1 [set] -> some expr2	expr2 some -> [set] expr1	expr2 some -> [set] expr1	expr1 [set] -> some expr2
expr1 lone -> one expr2	expr2 one -> lone expr1	expr2 one -> lone expr1	expr1 lone -> one expr2
expr1 lone -> some expr2	expr2 some -> lone expr1	expr2 some -> lone expr1	expr1 lone -> some expr2
expr1 one -> some expr2	expr2 some -> one expr1	expr2 some -> one expr1	expr1 one -> some expr2
expr1 in expr2	expr1 !in expr2	expr1 !in expr2	expr1 in expr2
expr1 = expr2	expr1 != expr2	expr1 != expr2	expr1 = expr2
expr1 + expr2	expr1 & expr2	expr1 & expr2	expr1 + expr2
expr1 + expr2	expr1 - expr2	expr1 - expr2	expr1 + expr2
expr1 & expr2	expr1 - expr2	expr1 - expr2	expr1 & expr2
expr1 - expr2	expr2 - expr1	expr2 - expr1	expr1 - expr2
expr1 in expr2	expr2 in expr1	expr2 in expr1	expr1 in expr2
expr1 !in expr2	expr2 !in expr1	expr2 !in expr1	expr1 !in expr2
expr1.expr2	expr2.expr1	expr2.expr1	expr1.expr2
expr1 <=> expr2	expr1 => expr2	expr1 => expr2	expr1 <=> expr2
expr1 => expr2	expr2 => expr1	expr2 => expr1	expr1 => expr2
expr1 <: expr2	expr1 :> expr2	expr1 :> expr2	expr1 <: expr2
expr1 ++ expr2	expr2 ++ expr1	expr2 ++ expr1	expr1 ++ expr2
expr1 ++ expr2	expr1 + expr2	expr1 + expr2	expr1 ++ expr2
expr1 < expr2	expr1 =< expr2	expr1 =< expr2	expr1 < expr2
expr1 < expr2	expr1 = expr2	expr1 = expr2	expr1 < expr2
expr1 < expr2	expr1 >= expr2	expr1 >= expr2	expr1 < expr2
expr1 < expr2	expr1 > expr2	expr1 > expr2	expr1 < expr2
expr1 =< expr2	expr1 = expr2	expr1 = expr2	expr1 =< expr2
expr1 =< expr2	expr1 >= expr2	expr1 >= expr2	expr1 =< expr2
expr1 =< expr2	expr1 > expr2	expr1 > expr2	expr1 =< expr2
expr1 = expr2	expr1 >= expr2	expr1 >= expr2	expr1 = expr2
expr1 = expr2	expr1 > expr2	expr1 > expr2	expr1 = expr2
expr1 >= expr2	expr1 > expr2	expr1 > expr2	expr1 >= expr2

or out of a **let** declaration. For example, the following **let** expression

let next = {n1: Node, n2: Node | n1 -> n2 **in** link}

{**all** n: Node | n **!in** n.^next}

can be modified as

let next = {n1: Node, n2: Node | n1 -> n2 **in** link}

{**all** n: Node | n **in** n.^next}

The above example is generated by applying mutation operator "**!in** to **in**" in the **let** declaration. Another example would be mutating the following statement

(**let** next = {n1: Node, n2: Node | n1 -> n2 **in** link}
{**all** n: Node | n **!in** n.^next}) => (**some** n: Node | **no** n.link)

to

(**some** n: Node | **no** n.link) =>
(**let** next = {n1: Node, n2: Node | n1 -> n2 **in** link}
{**all** n: Node | n **!in** n.^next})

The second example involves mutation operator "*expr1* => *expr2*" to "*expr2* => *expr1*".

4.1.9 If-Then-Else Expression

The **if-then-else** conditional expression takes the form

expr ::= expr (=> | **implies**) expr **else** expr

In the expression

b **implies** e1 **else** e2

b must be a boolean expression, and the result is the value of the expression *e1* when *b* evaluates to true and the value of *e2* when b evaluates to false.

Table 4.7: Mutate ITE Expression

From	*To*	*From*	*To*
expr1 => expr2 **else** expr3	expr1 => expr3 **else** expr2	expr1 => expr3 **else** expr2	expr1 => expr2 **else** expr3

There is only one mutation operator related to **if-then-else expression** as is shown in Table 4.7.

4.2 Mutation algorithm

This section describes the μAlloy algorithm for applying mutation operators to a given Alloy model to create its mutants. The **module** declaration or **import** statement of the model are ignored by μAlloy. The remaining 6 kinds of declarations in Alloy 4 are the subject of mutation:

- A **signature** declaration contains the following information:
  - Whether it is an **abstract** signature;
  - The multiplicity of the signature;
  - A keyword **sig**;
  - A signature name;
  - Whether it **extends** another signature;
  - A list of relation declarations; and
  - An optional block that restricts relations declared in the signature.
- A **fact** declaration has:

  - A keyword **fact**;
  - A name that may or may not be declared explicitly; and
  - A block that defines formulas that always hold.

- A **predicate** declaration consists of:
  - A keyword **pred**;
  - A predicate name;
  - Optionally a list of parameters; and
  - A block that contains formulas which must hold if the predicate is invoked.

- A **function** declaration is composed of:
  - A keyword **fun**;
  - A function name;
  - Optionally a list of parameters;
  - A return type; and
  - An expression invoked by providing an expression for each parameter.

- An **assert** declaration provides the following information:
  - A keyword **assert**;
  - A name that may or may not be declared explicitly; and

  - A block that contains properties of a model which can be later checked in a given scope.

- A **command** declaration contains:
  - Optionally a command name;
  - Keywords **run** or **check**;
  - A predicate/assertion name or a in-lined predicate/assertion declaration which is needed to be invoked; and
  - A scope under which the command is executed.

The mutation operators defined in section 4.1 apply to all 6 kinds of declarations but not necessarily to each component of each declaration. Table 4.8 lists the components to which the mutation operators apply.

Table 4.8: Parts of declarations to which mutation operations apply

Declaration	*Components where mutation operations apply*
signature	The Multiplicity of the signature
	Relation declarations
	The body of signature
fact	The body of fact
predicate	The body of predicate
function	The function return type
	The body of function
assert	The body of the assertion
command	In-lined predicate/assertion declaration (optional)
	The scope of command

Since different Alloy declarations do not share the same structure, μAlloy handle those declarations separately. In the following subsections, we

Input: Alloy .als file path, Directory path to which mutants are generated, Mutation operators
Output: Valid Alloy mutants

```
1  String readFrom ← alloy .als file path
2  String writeTo ← directory path to which mutants are generated
3  String[] mutationOp ← mutation operators that users provide
4  String[] fileMatrix ← readAlloyFile(readFrom)
5  AlloyAST module ← parseAlloyFile(readFrom)
6  if hasSigDecl(module) then
7  |  mutateSigDecls(fileMatrix, module, mutationOp, writeTo)
8  end
9  if hasFuncDecl(module) then
10 |  mutateFuncDecls(fileMatrix, module, mutationOp, writeTo)
11 end
12 if hasPredDecl(module) then
13 |  mutatePredDecls(fileMatrix, module, mutationOp, writeTo)
14 end
15 if hasFactDecl(module) then
16 |  mutateFactDecls(fileMatrix, module, mutationOp, writeTo)
17 end
18 if hasAssertDecl(module) then
19 |  mutateAssertDecls(fileMatrix, module, mutationOp, writeTo)
20 end
21 if hasCmdDecl(module) then
22 |  mutateCmdDecls(fileMatrix, module, mutationOp, writeTo)
23 end
```

Algorithm 1: Procedure *Main* for generating mutants

introduce the top level algorithm as well as algorithms for all declarations.

4.2.1 Top level algorithm

Algorithm 1 shows how μAlloy generates Alloy mutants. Section 5.3 shows how users can customize mutant generation.

In our algorithm, the user provides an Alloy file path to read from, a directory path to generate mutants and a set of mutation operators as inputs for μAlloy. μAlloy can generate valid mutants of a given Alloy model under the user given directory. In the algorithm, μAlloy first reads an Alloy model into a string array where each element corresponds to a unique line of the model, and then uses the information of the Alloy parse tree built by the compiler to mutate a specific part of the model. Each AST node in the Alloy parse tree contains location information corresponding to a token in the model, which facilitates the process of mutating Alloy models and generating mutants. μAlloy tries to visit each AST node in every existing Alloy declaration to match up with the user given mutation operators, and modifying the model to generate mutants. The modification update is made to the string array and the original file is left intact. After a mutation is made, μAlloy takes a snapshot of the string array and outputs the string array as a mutant under the user given directory. The tool then rollbacks the mutation operation on the string array and checks if the newly generated mutant is valid. If the mutant is invalid, then μAlloy simply deletes that mutant.

4.2.2 Signature

Algorithm 2 shows how μAlloy generates mutants while iterating through AST nodes in signature declarations (Algorithm 1 Line 7).

μAlloy iterates through all signature declarations and no mutant will be generated if no signatures is found. For each existing signature declaration,

Input: String[] *fileMatrix*, AlloyAST *module*, String[] *mutationOp*, String *writeTo*
Output: Valid Alloy mutants

```
1  foreach Signature sig ∈ getAllSignatures(module) do
2      fileMatrix ← mutateSigMult(mutationOp, sig)
3      Mutant muSig ← generateMutant(writeTo, fileMatrix)
4      if isInvalidMutant(muSig) then
5          deleteMutant(muSig)
6      end
7      rollBack(fileMatrix)
8      foreach Field field ∈ getAllFields(sig) do
9          fileMatrix ← mutateSigOp(mutationOp, field)
10         Mutant muSig ← generateMutant(writeTo, fileMatrix)
11         if isInvalidMutant(muSig) then
12             deleteMutant(muSig)
13         end
14         rollBack(fileMatrix)
15     end
16     if hasFact(sig) then
17         fileMatrix ← mutateFactOp(sig)
18         Mutant muFact ← generateMutant(writeTo, fileMatrix)
19         if isInvalidMutant(muFact) then
20             deleteMutant(muFact)
21         end
22         rollBack(fileMatrix)
23     end
24 end
```

Algorithm 2: Procedure *mutateSigDecl* for generating mutants

μAlloy mutates signature multiplicity based on the user given mutation operators. The tool also mutates unary operators or binary operators in every field/relation of signature declarations and generates mutants. For example, Figure 2.1 line 4 contains a unary operator "**lone**" and it can be replaced with another unary operator "**one**". Moreover, the field declaration "*link: State*

-> Node" can be modified to "*link: State* ***one -> some*** *Node*," where binary operator "**->**" is replaced with "**one->some**". If a signature declaration has a fact as is shown in Figure 2.1 lines 10-12, μAlloy will also mutate the fact declaration and generate mutants. For example, unary operator "**lone**" in Figure 2.1 line 11 can be replaced with "**one**", given the mutation operator "**lone** to **one**". All invalid mutants will be deleted once they are generated. After a mutant is generated, the algorithm will rollback to the state before applying any mutation operation. The process of checking mutant validity and deleting invalid mutant repeats each time a mutant is generated.

4.2.3 Function

This subsection illustrates the algorithm shown in Algorithm 1 line 10. The algorithm explains the process of mutating function declarations.

In Algorithm 3, muAlloy finds the return type of a function and applies mutation operators to it. For example, function

fun allListNodes(l: List): **set** Node{ l.header.*link }

has return type "***set*** *Node*" and it can be modified to "***one*** *Node*" or "***lone*** *Node*", etc. After mutating return type, μAlloy tries to mutate the body of the function. For example, the body of the above function declaration can be modified to

fun allListNodes(l: List): **set** Node{ l.header.^link }

where the unary operator "*" is replaced with "^".

Input: String[] *fileMatrix*, AlloyAST *module*, String[] *mutationOp*, String *writeTo*
Output: Valid Alloy mutants

```
1  foreach Function fun ∈ getAllFunctions(module) do
2  |  ReturnType retType ← findReturnType(fun)
3  |  fileMatrix ← mutateFuncOp(mutationOp, retType)
4  |  Mutant muFunc ← generateMutant(writeTo, fileMatrix)
5  |  if isInvalidMutant(muFunc) then
6  |  |  deleteMutant(muFunc)
7  |  end
8  |  rollBack(fileMatrix)
9  |  FunctionBody body ← findBody(fun)
10 |  while !visitAllSubnodes(body) do
11 |  |  ASTNode current ← getNextUnvisitedNode(body)
   |  |  fileMatrix ← mutateFuncOp(mutationOp, current)
12 |  |  Mutant muFunc ← generateMutant(writeTo, fileMatrix)
13 |  |  if isInvalidMutant(muFunc) then
14 |  |  |  deleteMutant(muFunc)
15 |  |  end
16 |  |  rollBack(fileMatrix)
17 |  end
18 end
```

Algorithm 3: Procedure *mutateFuncDecl* for generating mutants

4.2.4 Predicate

Algorithm 4 shows more details for line 13 in Algorithm 1.

To mutate Alloy predicate, muAlloy skips visiting AST nodes corresponding to the predicate parameters. Instead, it directly locates and accesses the body node for mutation. By traversing each sub-node under the body node, the tool performs mutations to the model by searching through every applicable mutation routines. If an opportunity of a mutation is found, μAlloy

Input: String[] *fileMatrix*, AlloyAST *module*, String[] *mutationOp*, String *writeTo*
Output: Valid Alloy mutants

```
1  foreach Predicate pred ∈ getAllPredicates(module) do
2  |  FunctionBody body ← findBody(pred)
3  |  while !visitAllSubnodes(body) do
4  |  |  ASTNode current ← getNextUnvisitedNode(body)
   |  |  fileMatrix ← mutatePredOp(mutationOp, current)
5  |  |  Mutant muPred ← generateMutant(writeTo, fileMatrix)
6  |  |  if isInvalidMutant(muPred) then
7  |  |  |  deleteMutant(muPred)
8  |  |  end
9  |  |  rollBack(fileMatrix)
10 |  end
11 end
```

Algorithm 4: Procedure *mutatePredDecl* for generating mutants

applies the mutation operator and generates a mutant.

4.2.5 Fact

Algorithm 5 illustrates more details for line 16 in Algorithm 1. It is very similar to Algorithm 4. The main differences include: (1) a predicate may have parameters while a fact does not; and (2) a fact may not be assigned a name explicitly, but a predicate must have a name. Since facts do not have parameter, it does not need a body node (The body node is conceptually equivalent to the fact node itself). Our tool simply iterates through all sub-nodes under the fact node and generates mutants under the guidance of mutation operators.

Input: String[] *fileMatrix*, AlloyAST *module*, String[] *mutationOp*, String *writeTo*
Output: Valid Alloy mutants

```
1  foreach Fact fact ∈ getAllFacts(module) do
2  |  while !visitAllSubnodes(fact) do
3  |  |  ASTNode current ← getNextUnvisitedNode(fact)
   |  |  fileMatrix ← mutateFactOp(mutationOp, current)
4  |  |  Mutant muPred ← generateMutant(writeTo, fileMatrix)
5  |  |  if isInvalidMutant(muPred) then
6  |  |  |  deleteMutant(muPred)
7  |  |  end
8  |  |  rollBack(fileMatrix)
9  |  end
10 end
```

Algorithm 5: Procedure *mutateFactDecl* for generating mutants

4.2.6 Assert

Algorithm 6 gives a brief description for line 19 in Algorithm 1. An assertion in Alloy may or may not have a name specified explicitly, and an unnamed assertion cannot be checked. Our tool iteratively goes through AST sub-nodes under assertions and detects opportunities to perform mutations.

4.2.7 Command

Algorithm 7 presents how μAlloy works when executing line 22 in Algorithm 1.

There is only one mutation operator defined for Alloy commands. In Algorithm 7 line 2, μAlloy only mutate the "***expect*** *Int*" part. For example, Figure 2.2 line 24 can be mutated to

Input: String[] *fileMatrix*, AlloyAST *module*, String[] *mutationOp*, String *writeTo*
Output: Valid Alloy mutants

```
1  foreach Assert assert ∈ getAllAsserts(module) do
2  |  while !visitAllSubnodes(assert) do
3  |  |  ASTNode current ← getNextUnvisitedNode(assert)
   |  |  fileMatrix ← mutateAssertOp(mutationOp, current)
4  |  |  Mutant muPred ← generateMutant(writeTo, fileMatrix)
5  |  |  if isInvalidMutant(muPred) then
6  |  |  |  deleteMutant(muPred)
7  |  |  end
8  |  |  rollBack(fileMatrix)
9  |  end
10 end
```

Algorithm 6: Procedure *mutateAssertDecl* for generating mutants

Input: String[] *fileMatrix*, AlloyAST *module*, String[] *mutationOp*, String *writeTo*
Output: Valid Alloy mutants

```
1  foreach Command cmd ∈ getAllCommands(module) do
2  |  fileMatrix ← mutateCmdExpect(mutationOp, cmd)
3  |  Mutant muCmd ← generateMutant(writeTo, fileMatrix)
4  |  if isInvalidMutant(muCmd) then
5  |  |  deleteMutant(muCmd)
6  |  end
7  |  rollBack(fileMatrix)
8  end
```

Algorithm 7: Procedure *mutateCmdDecl* for generating mutants

run RepOk **for exactly** 1 BinaryTree, 5 Node **expect** 0

In some cases, commands **run** or **check** may not explicitly invoke a named predicate or assertion. For example, line 26 in Figure 2.1 can be equivalently rephrased as

check {**all** l: List | RepOk[l] <=> RepOk2[l]} **for** 3 **but** 1 List

In such a case, we also need to mutate the predicate/assertion in the command declaration. The Alloy compiler parses those unnamed predicates/assertions in the same way as those named ones. Thus, all in-lined predicates and assertions in Alloy commands are modified in Algorithm 5 and Algorithm 6, so it is unnecessary to consider mutating unnamed predicates and assertions in Algorithm 7.

Chapter 5

Implementation and Evaluation

This chapter presents some implementation-level details of our prototype μAlloy and experimental results of applying it to 5 subject Alloy models. μAlloy traverses the parse tree built by Alloy analyzer; Section 5.1 describes these parse trees. Section 5.2 describes how we implement the mutation operators. Section 5.3 describes the GUI-based front-end of μAlloy. Section 5.4 presents some limitations of our prototype. Finally, Section 5.5 describes the experimental results.

5.1 Parse Tree

μAlloy operates on parse trees that Alloy analyzer builds. An Alloy parse tree is mainly composed of a set of Alloy AST nodes. The root of an Alloy parse tree is a node representing an Alloy module. A module contains a set of sub-nodes for Alloy constructs, including signatures, functions, predicates, facts and commands.

Figure 5.1 presents the Alloy parse tree for the singly linked list model in Figure 2.1. The root of the tree (**module**) contains the file path of the Alloy model. An Alloy module imports the built-in *integer* module by default,

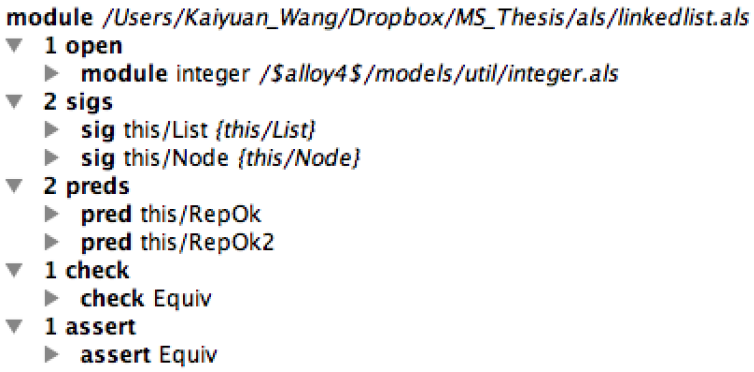


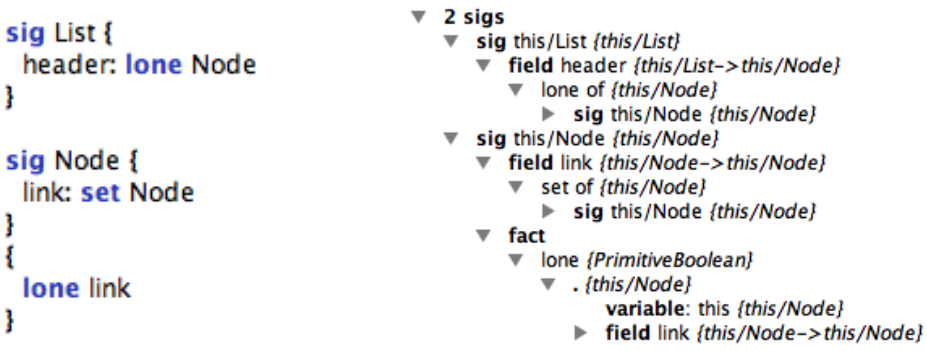
Figure 5.1: Parse tree for singly linked list

as is shown under AST node **open**. The *integer* module defines basic integer constants and integer operations for Alloy models. The parse tree shows that the singly linked list Alloy model has:

- 2 signature declarations (**sig** *List* and **sig** *Node*);
- 2 predicate declarations (**pred** *RepOk* and **pred** *RepOk2*);
- 1 assertion declaration (**assert** Equiv); and
- 1 check command declaration (**check** Equiv).

Each AST node in Figure 5.1 can be expanded further to expose more details of the model.

Figure 5.2 shows the signature declarations and the corresponding parse tree. The parse tree contains the necessary information for mutating signature declarations, which includes:



(a) Signature declaration

(b) Signature parse tree

Figure 5.2: Expanded Alloy parse tree for signatures in singly linked list

- A set of subtrees including AST nodes which represent all signature declarations;
- The signature multiplicity (not shown in Figure 5.2b) for each signature declaration;
- Cardinalities for relations defined in each signature;
- Constraints for relations represented by the optional **fact** node under each signature; and
- The location information for each AST node.

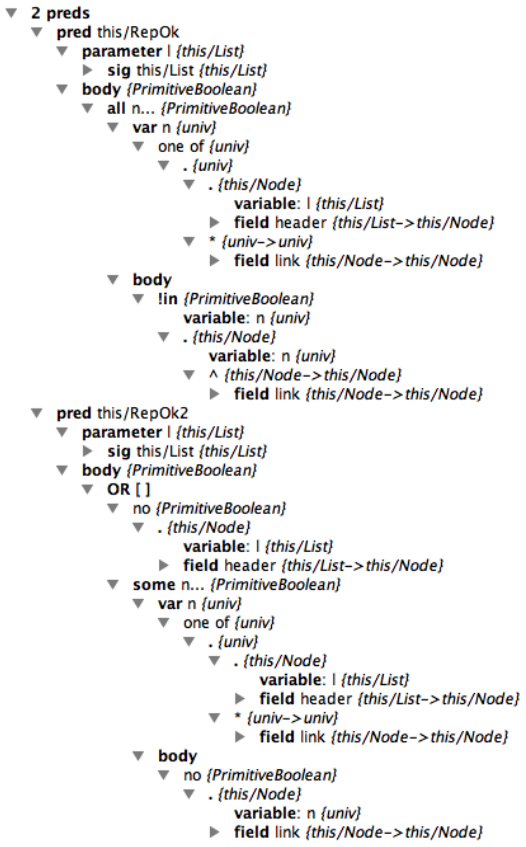
Figure 5.3b shows the parse tree for predicate declarations in the singly linked list model. It contains the information in Figure 5.3a necessary for mutations, including:

- A set of sub-trees containing AST nodes that represent all predicate declarations;

```
pred RepOk(l: List) {
  all n: l.header.*link | n !in n.^link
}

pred RepOk2(l: List) {
  no l.header || some n: l.header.*link | no n.link
}
```

(a) Predicate declaration



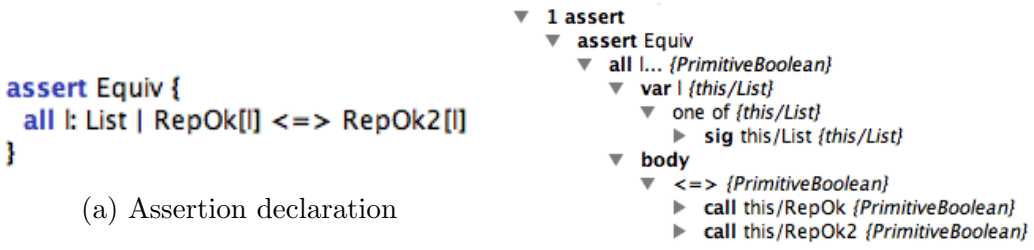
(b) Predicate parse tree

Figure 5.3: Expanded Alloy parse tree for predicates in singly linked list

- Formulas in each predicate; and
- The location information for each AST node.

Figure 5.4b is the parse tree for the assertion declaration in the singly linked list model. It contains the following information used by μAlloy:

- A sub-tree involving AST nodes that represent the assertion declaration;



(a) Assertion declaration

(b) Assertion parse tree

Figure 5.4: Expanded Alloy parse tree for assertions in singly linked list

- Formulas in each assertion; and
- The location information for each AST node.

Figure 5.5 shows the Alloy parse tree for the binary tree model in Figure 2.2. The parse tree shows the structure of the binary tree model and contains:

- 2 signature declarations (**sig** *BinaryTree* and **sig** *Node*);
- 1 predicate declaration (**pred** *RepOk*);
- 1 fact declaration (**fact** *fact$1*); and
- 1 run command declaration (**run** *RepOk*).

Each AST node in Figure 5.5 can be unrolled further to show more AST nodes of the binary tree model. The subtree for the signature declarations is shown in Figure 5.6b. It contains necessary information for mutation similar to what we discussed for signature declarations in the singly linked list model.

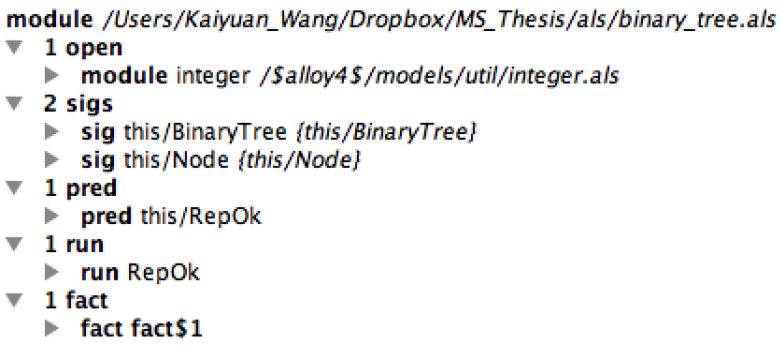
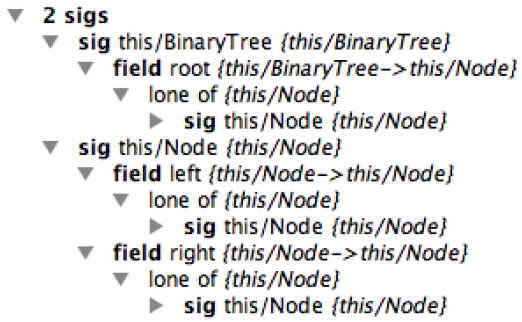


Figure 5.5: Parse tree for binary tree

```
one sig BinaryTree {
  root: lone Node
}

sig Node {
  left: lone Node,
  right: lone Node
}
```

(a) Signature declaration



(b) Signature parse tree

Figure 5.6: Expanded Alloy parse tree for signatures in binary tree

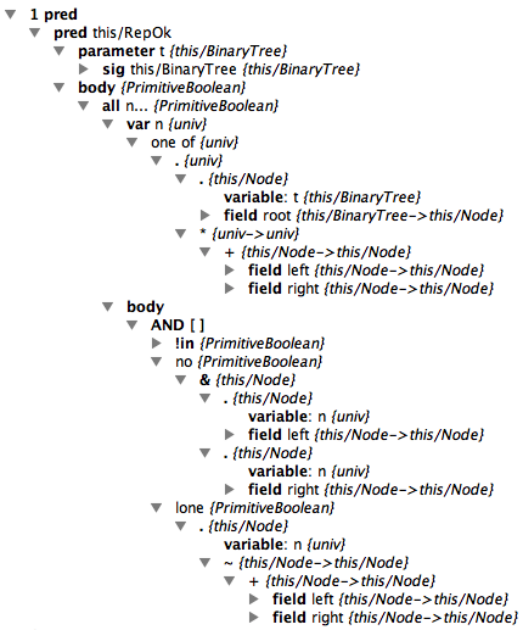
Figure 5.7b presents the parse tree for the predicate declaration in Figure 5.7a. It contains necessary information for μAlloy to perform mutations.

Figure 5.8b is the parse tree corresponding to the unnamed **fact** declaration in Figure 5.8a. The tree contains necessary information for mutant generation, including:

- A set of AST nodes that represent constraints that always hold; and
- The location information for each AST node.

```
pred RepOk(t: BinaryTree) {
  all n: t.root.*(left + right) {
    n !in n.^(left + right)
    no n.left & n.right
    lone n.~(left + right)
  }
}
```

(a) Predicate declaration

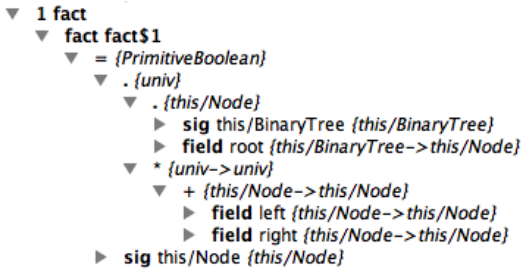


(b) Predicate parse tree

Figure 5.7: Expanded Alloy parse tree for predicates in binary tree

```
fact {
  BinaryTree.root.*(left + right) = Node
}
```

(a) Fact declaration



(b) Fact parse tree

Figure 5.8: Expanded Alloy parse tree for facts in binary tree

μAlloy iteratively traverses every AST node and invokes the corresponding mutation routine. When a routine recognizes an opportunity for a mutation, it creates a mutant. While the Alloy parse tree is used to generate mutants, it cannot be directly used for our tool in the sense that (1) the parse

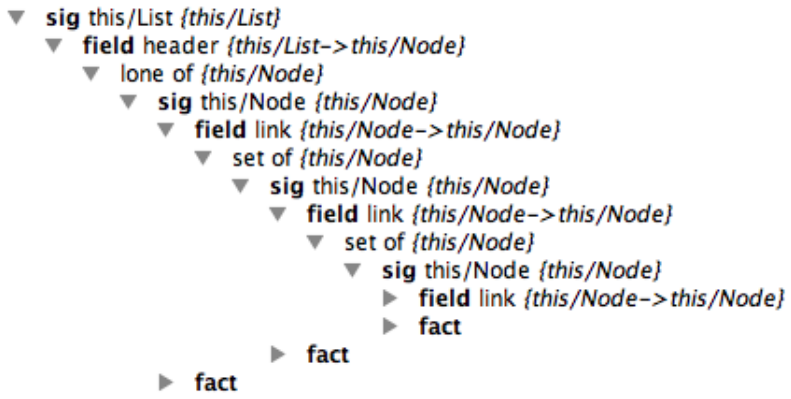


Figure 5.9: Parse tree unrolling that shows cycles

tree can have cycles and the AST nodes corresponding to signatures can be unrolled indefinitely (Figure 5.9); and (2) it does not store all information for Alloy models, e.g. location information for parentheses and keyword **let**, etc., is missing. μAlloy handles these cases appropriately.

5.2 Mutation Operator Implementation

Mutation operators for Alloy are implemented in groups based on their structure similarities. In this way, we don't have to write a function for each mutation operator, rather we build a function for each group of mutation. The following subsections explain how μAlloy covers our mutation operators for Alloy.

5.2.1 Number increment and decrement

μAlloy implements mutation operators shown in Table 4.2 for integers. Integer nodes in Alloy parse tree are stored in the type of *ExprConstant*. Table 5.1 presents the only operator μAlloy supports in class *ExprConstant*.

Table 5.1: Operators in ExprConstant

Operator	*Description*
NUMBER("NUMBER")	An integer constant

When μAlloy detects an integer *ExprConstant*, say integer i, it accesses location information of i from within the corresponding AST node whose run-time class is *ExprConstant* and replaces i with $i + 1$ or $i - 1$.

5.2.2 Unary operator mutation

For unary operators, μAlloy supports mutation operators shown in Table 4.3 and Table 4.5. These mutation operators include multiplicity replacements, unary operator replacements, unary operator insertion and unary operator deletion.

Signature multiplicities are stored in signature declaration nodes, which are of class *Sig*. μAlloy accesses related fields in the *Sig* nodes and if the tool recognizes a chance for multiplicity replacement mutation, it applies the mutation operator and generates a mutant.

Unary operators in Alloy parse tree are stored in *ExprUnary* class. Table 5.2 shows the operators μAlloy supports in class *ExprUnary*. μAlloy iterates through all AST nodes and searches for the matching operators in

Table 5.2: Operators in ExprUnary

Operator	*Description*
SOMEOF("some of")	:some x (where x is a unary set)
LONEOF("lone of")	:lone x (where x is a unary set)
ONEOF("one of")	:one x (where x is a unary set)
SETOF("set of")	:set x (where x is a unary set)
NOT("!")	not f (where f is a formula)
NO("no")	no x (where x is a set or relation)
SOME("some")	some x (where x is a set or relation)
LONE("lone")	lone x (where x is a set or relation)
ONE("one")	one x (where x is a set or relation)
TRANSPOSE("~")	transpose
RCLOSURE("*")	reflexive closure
CLOSURE("^")	closure

Table 5.2 to: (1) perform unary operator replacements; (2) insert unary operators; or (3) delete unary operators.

5.2.3 Quantifier mutation

μAlloy supports quantifier related mutation operators which are listed in Table 4.4. These mutation operators only involve quantifier replacements.

Quantifiers in Alloy parse tree are stored in *ExprQt* class. Table 5.3 shows the quantifiers μAlloy supports in class *ExprQt*.

Table 5.3: Operators in ExprQt

Operator	*Description*
ALL("all")	all a,b:x | formula
NO("no")	no a,b:x | formula
LONE("lone")	lone a,b:x | formula
ONE("one")	one a,b:x | formula
SOME("some")	some a,b:x | formula

5.2.4 Binary operator mutation

Mutation operators involving binary operators shown in Table 4.6 are supported in μAlloy. These operators include: (1) binary operator replacements; and (2) swapping left subexpression and right subexpression of binary operators.

Binary operators in Alloy are stored in *ExprBinary* class. Table 5.4 shows the binary operators that μAlloy supports in *ExprBinary* class. μAlloy traverses through Alloy parse trees and detects the binary operators in Table 5.4 to perform mutations.

5.2.5 If-Then-Else operator mutation

μAlloy supports the mutation operator shown in Table 4.7, which involves swapping **then** clause and **else** clause of the **if-then-else** expression.

The **if-then-else** expression in Alloy parse tree is of class *ExprITE*. The *ExprITE* class stores the *if* condition formula, the **then** clause and the **else** clause. μAlloy goes through Alloy parse trees and detects **if-then-else** expressions for mutations. Whenever a node whose runtime class is *ExprITE* is detected, μAlloy swaps its **then** clause and **else** clause to generate a mutant.

5.3 GUI front-end for μAlloy

We implement a simple GUI front-end for μAlloy to allow users to more easily customize mutation operators. Figure 5.10 shows the key parts of the

Table 5.4: Operators in ExprBinary

Operator	*Description*
ARROW(“->”)	cross product
ANY_ARROW_SOME(“->some”)	cross product
ANY_ARROW_ONE(“->one”)	cross product
ANY_ARROW_LONE(“->lone”)	cross product
SOME_ARROW_ANY(“some->”)	cross product
SOME_ARROW_SOME(“some->some”)	cross product
SOME_ARROW_ONE(“some->one”)	cross product
SOME_ARROW_LONE(“some->lone”)	cross product
ONE_ARROW_ANY(“one->any”)	cross product
ONE_ARROW_SOME(“one->some”)	cross product
ONE_ARROW_ONE(“one->one”)	cross product
ONE_ARROW_LONE(“one->lone”)	cross product
LONE_ARROW_ANY(“lone->”)	cross product
LONE_ARROW_SOME(“lone->some”)	cross product
LONE_ARROW_ONE(“lone->one”)	cross product
LONE_ARROW_LONE(“lone->lone”)	cross product
JOIN(“.”)	relational join
DOMAIN(“<:”)	domain restriction
RANGE(“:>”)	range restriction
INTERSECT(“&”)	intersection
PLUSPLUS(“++”)	override
PLUS(“+”)	union
MINUS(“-”)	difference
EQUALS(“=”)	equals
NOT_EQUALS(“!=”)	not equals
IMPLIES(“=>”)	implies
LT(“<”)	less than
LTE(“<=”)	less than or equal
GT(“>”)	greater than
GTE(“>=”)	greater than or equal
NOT_LT(“!<”)	not less than
NOT_LTE(“!<=”)	not less than or equal
NOT_GT(“!>”)	not greater than
NOT_GTE(“!>=”)	not greater than or equal
IN(“in”)	subset
NOT_IN(“!in”)	not subset
IFF(“<=>”)	bi-imply

μAlloy GUI.

GUI μAlloy allows users to provide mutation operators and select Alloy declarations that mutation operators apply to. For the settings shown in Figure 5.10, users select signatures, predicates, facts, check commands and run commands to apply mutation operators.

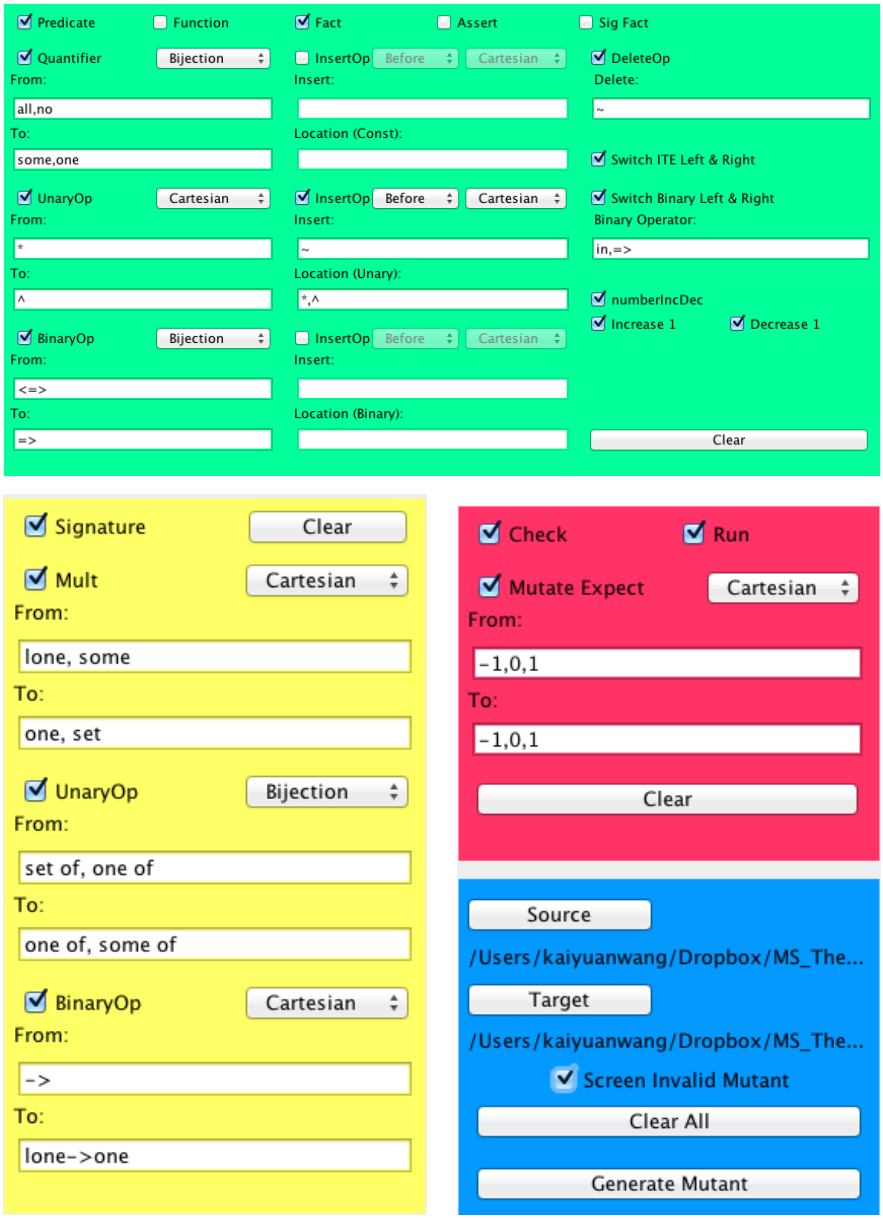


Figure 5.10: GUI front-end for μAlloy

Table 5.5: Mutation operators for signatures under the settings in Figure 5.10

Related to	*Mutation operator*
Multiplicity replacement	**lone** to **one**
	lone to **set**
	some to **one**
	some to **set**
Unary operator replacement	**set of** to **one of**
	one of to **some of**
Binary operator replacement	`->` to **lone->one**

In the yellow panel, users can customize settings to modify signature declarations and generate mutants. GUI μAlloy allows users to provide mutation operators that modify signature multiplicities and relation cardinalities. Table 5.5 shows the mutation operators provided by the user in the yellow panel of Figure 5.10.

The combo box "Cartesian" next to check box "Mult" helps define the mutation operators for signature multiplicity in the way that the sets of operators in text fields "From" and "To" are combined as Cartesian product. For example, the set of operators "**lone**" and "**some**" in text field "From" and the set of operators "**one**" and "**set**" in text field "To" are combined to create 4 mutation operators for signature multiplicities. On the other hand, the combo box "Bijection" next to check box "UnaryOp" shows that mutation operators related to unary operators for relation cardinalities are defined in the way that the sets of operators in text field "From" and "To" are combined as ordered bijection relation. For example, the set of operators "**set of**" and "**one of**" in text field "From" and the set of operators "**one of**" and "**some**

Table 5.6: Mutation operators for formulas under the settings in Figure 5.10

Related to	*Mutation operator*
Quantifier replacement	**all -> some**
	no -> one
Unary operator replacement	*** -> ^**
Binary operator replacement	**<=> -> =>**
Insert unary operator	~ before *****
	~ before ^
Delete unary operator	~
Swap ITE clauses	a **=>** b **else** c **->** a **=>** c **else** b
Swap binary operator subexpressions	a **binOp** b **->** b **binOp** a
Number inc/dec	**num -> num +** 1
	num -> num - 1

of" in text field "To" are joined to create 2 mutation operators shown in Table 5.5.

In the green panel, users customize settings to alter predicates, functions, facts, asserts and signature facts. GUI μAlloy allows users to define mutation operators related to quantifiers, unary operators, binary operators, integers, etc. Table 5.6 shows the mutation operators provided in the green panel of Figure 5.10.

The settings in the green panel is similar to what we've explained above, so it's quite easy to understand what the settings in the green panel mean.

In the red panel, users can choose to apply mutations on **check** commands or **run** commands. The only mutation operators designed for Alloy command is mutating the "***expect*** *Int*" part of the commands as is shown in Table 5.7.

Table 5.7: Mutation operators for command under the settings in Figure 5.10

Related to	*Mutation operator*
Command expect	... -> ...**expect 0**
	... -> ...**expect int** (int > 0)
	...**expect 0** -> ...
	...**expect 0** -> ...**expect int** (int > 0)
	...**expect int** (int > 0) -> ...
	...**expect int** (int > 0) -> ...**expect 0**

5.4 Limitations

Our current prototype has the following limitations:

- No more than one signature declaration can appear on one line;

  For example, signature declarations

  ```
  sig A {} sig B {}
  ```

  may not be used, however, the following equivalent signature declarations works:

  ```
  sig A {}
  sig B {}
  ```

  The reason for this problem is that Alloy parse tree does not store location information for keyword **sig**. So μAlloy is designed to locate the keyword **sig** by string manipulation, which returns the location where the first string "**sig**" appears. If two or more keywords **sig** are in the same line, μAlloy could behave incorrectly.

- An expression may not be declared in multiple lines of code;

  For example, μAlloy supports performing mutations for expressions written in the same line, such as

  ```
  a + b
  ```

  However, the semantically equivalent expression

  ```
  a
  +
  b
  ```

  may not be handled properly by the current version of μAlloy, although the expression is also valid and can be interpreted by Alloy analyzer.

  The reason for this problem is that current version of μAlloy only supports swapping left sub-expression and right sub-expression within the same line.

- If a **let** expression *expr* is involved in a mutation, other keywords **let** are not allowed to be written in the same line with the keyword **let** of *expr*. μAlloy may not be able to correctly handle cases where two or more keywords **let** appear in the same line of code.

  The reason is that Alloy parse trees do not store location information for keyword **let**, so μAlloy is designed to locate the keyword **let** simply by searching through the string of the line.

Table 5.8: μAlloy experiment results

Model	*#Total mutants*	*#Valid mutants*	*#Invalid mutants*	*Time [sec]*
Singly linked list	173	NA	NA	< 0.1
Binary tree	250	NA	NA	< 0.1
Boolean	148	NA	NA	< 0.1
FSM	951	NA	NA	0.2
Class diagram	1667	NA	NA	0.3

5.5 Experiments

We apply μAlloy to 5 subject Alloy models to observe the number of mutants generated as well as time taken for generation. The subject models include the singly linked list model (Fig 2.1), the binary tree model (Fig 2.2), the boolean model (Fig A.1), the finite state machine model (Fig A.2) and the class diagram model (Fig A.3).

The experiments were run on a MacBook Pro with 2.5GHz quad-core Intel Core i7 turbo boost up to 3.7 GHz. The operating system is OS X Yosemite version 10.10.3. Table 5.8 shows the results for mutant generation when invalid mutant screening is turned off. The number of mutants generated ranges from 173 to 1667. In all cases mutant generation takes less than 1 second.

Table 5.9 shows the results for mutant generation when invalid mutant screening is turned on and μAlloy analyzes each candidate mutant and rejects the invalid ones. The number of valid mutants generated ranges from 72 to 351. The generation time ranges from less than 1 second to 12.2 seconds.

Table 5.9: μAlloy experiment results

Model	*#Total mutants*	*#Valid mutants*	*#Invalid mutants*	*Time [sec]*
Singly linked list	173	72	101	0.8
Binary tree	250	88	162	1.1
Boolean	148	68	80	0.7
FSM	951	212	739	6.0
Class diagram	1667	351	1316	12.2

Chapter 6

Potential Applications

This chapter describes two techniques, one for automated testing and one for automated debugging, based on the foundation laid by μAlloy:

- Mutation testing for Alloy models (Section 6.1); and
- Repairing buggy Alloy models (Section 6.2).

6.1 Mutation testing Alloy models

We define *mutation testing* of declarative Alloy models in the spirit of mutation testing of imperative programs by leveraging an existing test automation framework AUnit, which introduces unit testing for Alloy. Specifically, AUnit defines test cases, test pass/fail evaluation, as well as code coverage for Alloy models. Thus, AUnit lays the basic groundwork for testing Alloy models. AUnit defines a test case as a pair <σ, ρ> where σ is either an assignment of values to some or all sets and relations declared in signatures, and ρ is an Alloy command. A test case <σ, ρ> passes if σ is a solution to the constraint-solving problem for the command ρ of the given model. Otherwise, it fails. More details on AUnit can be found elsewhere [24].

```
1  module list
2
3  sig Node {
4    link: set Node
5  }
6
7  fact PartialFunction {
8    all n: Node | lone n.link
9  }
10
11 run {}
```

Figure 6.1: Example model for mutation testing

We follow the spirit of traditional mutation testing to define *mutant killing* for Alloy: if a test case t passes when executing ρ on a model m, but fails when executing ρ on m's mutant m', we say the test t kills the mutant m'. Recall the mutation score for a test suite is the number of mutants killed divided by the total number of mutants generated.

Figure 6.1 presents our example Alloy model, which models singly-linked lists (which may or may not have cycles). The model has a *fact* which restricts each node to connect to zero or one node through the relation *link*.

μAlloy generates 17 valid mutants for the list model in Figure 6.1.

Table 6.1: Test suite for the list model in Figure 6.1

Command	*Visualized valuation*
run {}	∅ empty instance
run {}	
run {}	
run {}	

```
1  module list
2
3  sig Node {
4    link: lone Node // mutation: "set" --> "lone"
5  }
6
7  fact PartialFunction {
8    all n: Node | lone n.link
9  }
10
11 run {}
```

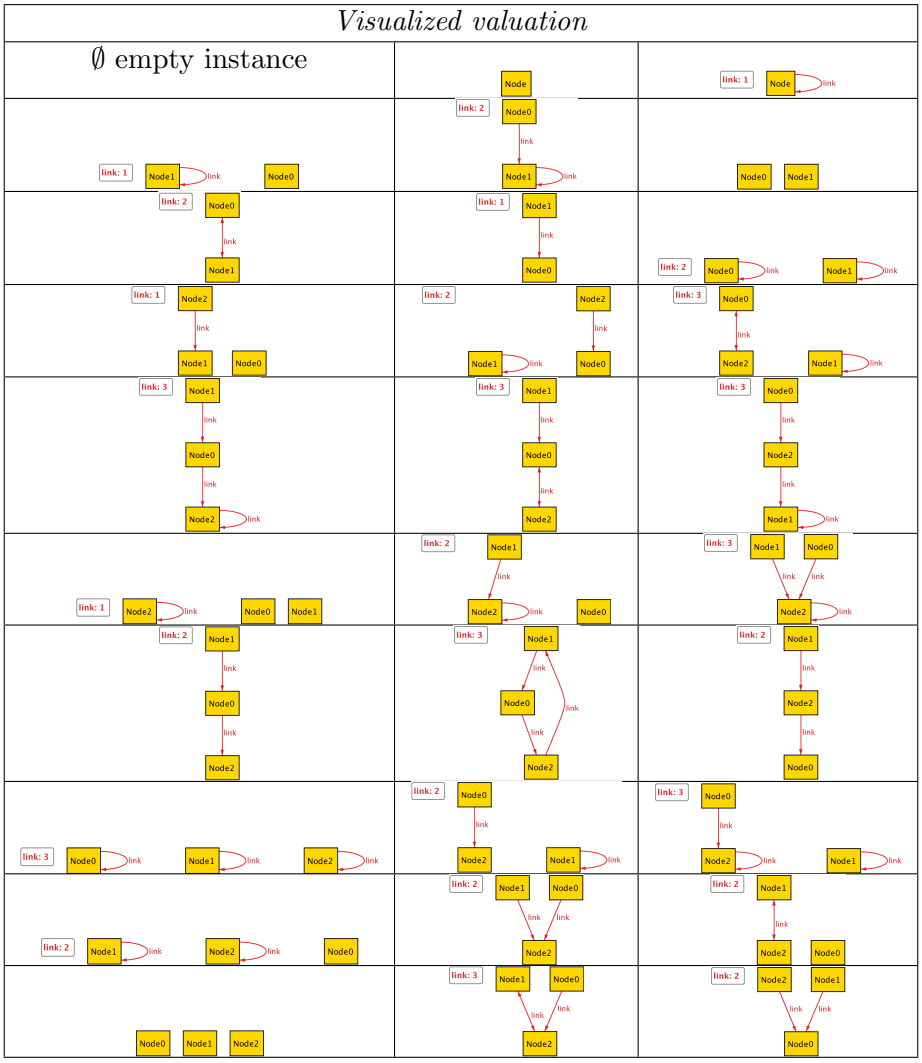
Figure 6.2: Example equivalent mutant

Table 6.1 presents an example test suite T with 4 test cases for the list model. Each of these tests passes for our initial model (Figure 6.1), i.e., each valuation shown is a valid instance of the model. We run each test case in T against each of the 17 mutants, and T kills 14 mutants. Thus, the mutation score for T is 0.82.

As another example, consider a larger test suite T', which contains *all* instances generated by the Alloy analyzer for our initial model (Figure 6.1). Thus, each test in T' is a tuple that has its first element the command "***run*** {}" and second element a satisfying valuation w.r.t. the constraints in the model. Table 6.2 shows all 30 instances generated by the analyzer. Each instance along with a command "***run*** {}" produces a test case. We run each of the 30 tests in T' against each of the 17 mutants, and T' kills 16 mutants, which gives a mutation score of 0.94.

The one mutant that is not killed by T' is shown in Figure 6.2. The mutant differs with the original model as follows. Line 4 is changed from

Table 6.2: Full test suite for the list model in Figure 6.1



"*link:* ***set*** *Node*" to "*link:* ***lone*** *Node*", thereby modifying the declaration of the field *link* to be a partial function instead of any arbitrary relation. Notice that the original model already includes a fact (*PartialFunction*) that restricts *link* to be a partial function. Thus, the mutant and the original model are

semantically equivalent and no test can kill this mutant.

Table 6.3 presents the details of mutant killing using the 30 test cases against the 17 mutants. An entry "K" in row t_i, column m_j means test case t_i kills mutant m_j. An entry "N" in row t_i, column m_j means test case t_i does not kill mutant m_j. An entry "–" in row t_i, column m_j means test case t_i was not run against mutant m_j since it was already killed by another test.

Table 6.3: 30 test cases vs 17 mutants killing chart

test case / mutant	m_1	m_2	m_3	m_4	m_5	m_6	m_7	m_8	m_9	m_{10}	m_{11}	m_{12}	m_{13}	m_{14}	m_{15}	m_{16}	m_{17}
t_1	K	K	K	N	K	K	N	N	N	N	N	N	K	K	N	K	N
t_2	–	–	–	K	–	–	N	N	N	N	N	N	–	–	N	–	N
t_3	–	–	–	–	–	–	K	N	N	N	K	N	–	–	K	–	K
t_4	–	–	–	–	–	–	–	K	N	K	–	N	–	–	–	–	–
t_5	–	–	–	–	–	–	–	–	N	–	–	N	–	–	–	–	–
t_6	–	–	–	–	–	–	–	–	K	–	–	N	–	–	–	–	–
t_7	–	–	–	–	–	–	–	–	–	–	–	N	–	–	–	–	–
t_8	–	–	–	–	–	–	–	–	–	–	–	N	–	–	–	–	–
t_9	–	–	–	–	–	–	–	–	–	–	–	N	–	–	–	–	–
t_{10}	–	–	–	–	–	–	–	–	–	–	–	N	–	–	–	–	–
t_{11}	–	–	–	–	–	–	–	–	–	–	–	N	–	–	–	–	–
t_{12}	–	–	–	–	–	–	–	–	–	–	–	N	–	–	–	–	–
t_{13}	–	–	–	–	–	–	–	–	–	–	–	N	–	–	–	–	–
t_{14}	–	–	–	–	–	–	–	–	–	–	–	N	–	–	–	–	–
t_{15}	–	–	–	–	–	–	–	–	–	–	–	N	–	–	–	–	–
t_{16}	–	–	–	–	–	–	–	–	–	–	–	N	–	–	–	–	–
t_{17}	–	–	–	–	–	–	–	–	–	–	–	N	–	–	–	–	–
t_{18}	–	–	–	–	–	–	–	–	–	–	–	N	–	–	–	–	–
t_{19}	–	–	–	–	–	–	–	–	–	–	–	N	–	–	–	–	–
t_{20}	–	–	–	–	–	–	–	–	–	–	–	N	–	–	–	–	–
t_{21}	–	–	–	–	–	–	–	–	–	–	–	N	–	–	–	–	–
t_{22}	–	–	–	–	–	–	–	–	–	–	–	N	–	–	–	–	–
t_{23}	–	–	–	–	–	–	–	–	–	–	–	N	–	–	–	–	–
t_{24}	–	–	–	–	–	–	–	–	–	–	–	N	–	–	–	–	–
t_{25}	–	–	–	–	–	–	–	–	–	–	–	N	–	–	–	–	–
t_{26}	–	–	–	–	–	–	–	–	–	–	–	N	–	–	–	–	–
t_{27}	–	–	–	–	–	–	–	–	–	–	–	N	–	–	–	–	–
t_{28}	–	–	–	–	–	–	–	–	–	–	–	N	–	–	–	–	–
t_{29}	–	–	–	–	–	–	–	–	–	–	–	N	–	–	–	–	–
t_{30}	–	–	–	–	–	–	–	–	–	–	–	N	–	–	–	–	–

6.2 Repair Alloy models

We next describe how μAlloy can provide an enabling technology for repairing buggy Alloy models. Our approach to repair follows the spirit of repairing imperative programs using given test suites [11, 13]. Specifically, given a test suite T and a faulty Alloy model m such that at least one test in T fails, our approach iteratively creates mutants of m and checks whether all the tests in T pass for any of the mutants. Any mutant that passes all the tests is identified as a potential fix, which the user inspects to keep or discard.

Figure 6.3 shows an example faulty Alloy model, which is intended to represent the following properties:

- There exists only one list;
- The list has at most one header;
- Each node has a link, which points to at most one node;
- All nodes are reachable from the header of the list;
- The list does not have a cycle; and
- The scope only allows 1 list and up to 2 nodes.

The field declaration for *header* erroneously over-constrains the relation to be a total function (Line 4); it should instead be declared as a partial function. In our experience, Alloy beginner users can make that mistake. It is also hard to find a bug like this in an Alloy model, since the bug does

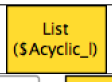
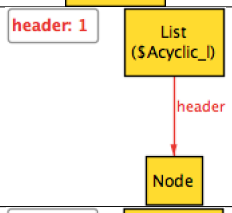
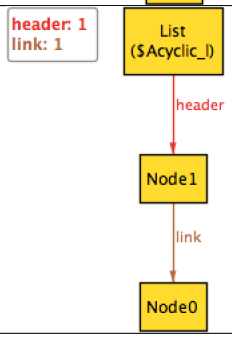
```
1  module linkedlist_bug
2
3  one sig List {
4    header: Node // fault here: should be ``header: set Node''
5  }
6
7  sig Node {
8    link: lone Node
9  }
10
11 fact {
12   lone header
13   List.header.*link = Node
14 }
15
16 pred Acyclic(l: List) {
17   all n: l.header.*link | n !in n.^link
18 }
19
20 run Acyclic for 2 but 1 List
```

Figure 6.3: Buggy singly linked list model

not produce unexpected instances; it only causes the analyzer to not generate certain expected valuations.

Table 6.4 shows an example test suite that the user provides to our repair algorithm; each test is expected to pass against a correct model. Test case 1 has an instance with 1 list with 0 headers and 0 node. Test case 2 has an instance with 1 list with 1 header and 1 node with 0 link. Test case 3 has an instance with 1 list with 1 header and 2 nodes with 1 link for *Node1* and 0 link for *Node0*. When run against the buggy model (Figure 6.3), Test 1 fails but the other two tests pass. Specifically, the execution of Test 1 leads to an unsatisfiable SAT formula.

Table 6.4: Test suite for the buggy list model in Figure 6.3.

Test case	*Command*	*Visualized valuation*	*Outcome*
1	**run** Acyclic for 2 but 1 List		Fail
2	**run** Acyclic for 2 but 1 List		Pass
3	**run** Acyclic for 2 but 1 List		Pass

μAlloy generates 50 valid mutants of the buggy singly linked list model (Figure 6.3). Figure 6.4 and Figure 6.5 illustrate the only two mutants that pass all the tests and are potential fixes shown to the user. Both these mutants are semantically equivalent. The user may choose either of them as the desired fix.

```
1  module linkedlist_bug
2
3  one sig List {
4    header: lone Node // repair from default (one) to lone
5  }
6
7  sig Node {
8    link: lone Node
9  }
10
11 fact {
12   lone header
13   List.header.*link = Node
14 }
15
16 pred Acyclic(l: List) {
17   all n: l.header.*link | n !in n.^link
18 }
19
20 run Acyclic for 2 but 1 List
```

Figure 6.4: Buggy singly linked list

```
1  module linkedlist_bug
2
3  one sig List {
4    header: set Node // repair from default (one) to set
5  }
6
7  sig Node {
8    link: lone Node
9  }
10
11 fact {
12   lone header
13   List.header.*link = Node
14 }
15
16 pred Acyclic(l: List) {
17   all n: l.header.*link | n !in n.^link
18 }
19
20 run Acyclic for 2 but 1 List
```

Figure 6.5: Buggy singly linked list

Chapter 7

Related Work

Mutation testing for imperative programs is a well-studied area[10, 12, 14, 21]. While the original application of mutation testing was in the context of test suite quality [10], more recent projects have explored other applications, specifically for mutation, e.g., for program repair [8]. In the context of evaluating test suite quality, mutation testing is powerful but expensive since in the worst case it can require running all tests against all mutants and the original program. Several techniques aim to lower the cost of mutation testing, for example using a subset of mutation operators or mutants [6, 15, 17, 19] or by integrating from regression testing with mutation testing [25].

Mutation testing for declarative programs is a lesser explored area [7]. Previous work in this area has focused largely on mutation of specifications (typically for imperative code), where specific techniques apply mutation operators designed for imperative code but also applicable to specifications, e.g., applying relational operator replacement to replace "<" with ">".

Our work is closest in spirit to Srivatanakul et al. [23] who define mutation operators for CSP specifications written using FDR2 syntax [2]. Specifically, their process definition operators focus on specific specification con-

structs. The key difference is our support for Alloy – a relational first-order logic with transitive closure.

Aichernig and Salas [3] define specification mutation for OCL and apply it to pre/post-condition specifications for constraint-based testing. The mutation operators for OCL are a subset of those used commonly for imperative code, specifically for boolean expression modification. A generalization and formalization of the foundation of this work is presented in follow up work by Aichernig and Jifeng [4] who provide an integration of mutation testing with the Unifying Theory of Programming [12].

Sullivan's masters thesis recently introduced AUnit [24], a unit testing framework for Alloy in the spirit of popular unit testing frameworks, such as JUnit, for imperative languages. As we show in Chapter 6, μAlloy integrates with AUnit to provide mutation testing for Alloy in the spirit of mutation testing for imperative code, where mutation score is a metric for test suite quality. We believe μAlloy with AUnit is the first such mutation testing framework for Alloy.

Besides μAlloy, a number of projects have developed Alloy extensions. KodKod [5] introduced a new SAT-based back-end for Alloy and provided an API to make it easier for other projects to benefit from the Alloy tool-set. Montaghami and Rayside [16] support direct specification of partial instances to allow users to fine tune the instance generation. Aluminum [18] introduces generation of minimal instances to ease instance inspection.

Chapter 8

Conclusion

This thesis introduced a foundation for mutation-driven analyses for Alloy, a first-order, declarative language based on relations. Specifically, we introduced a family of mutation operators for Alloy models and defined algorithms for applying the operators on different parts of the models. Our prototype tool μAlloy embodies these operators and algorithms and provides a GUI-based front-end for customizing the application of mutation operators. To demonstrate the potential of our approach, we illustrated the use of μAlloy in two application scenarios: (1) mutation testing for Alloy (in the spirit of traditional mutation testing for imperative languages); and (2) program repair for Alloy using mutation. We believe mutation testing has a valuable role to play in developing correct models in Alloy as well as other similar declarative languages.

Appendix

Appendix A

Alloy Model Appendix

A.1 Boolean

The *Boolean* model in Figure A.1 defines **Boolean** type for Alloy models. A **Boolean** variable in Alloy can either be **True** or **False**. The model also defines basic boolean operations for any given two boolean variables.

A.2 Finite State Machine

The *Finite State Machine* model in Figure A.2 defines signature *Node* and *Transition* for finite state machine, along with constraints to which every instance of FSM complies.

A.3 Java Class Diagram

Figure A.3 shows the definition of Java class diagram in Alloy. The model defines basic components for class diagrams such as ***sig*** *Class*, ***sig*** *Interface* and ***sig*** *Association*. It also declares constraints that restrict class diagrams for Java programs.

```
1  module boolean
2
3  abstract sig Bool {}
4  one sig True, False extends Bool {}
5
6  pred isTrue[b: Bool] { b in True }
7
8  pred isFalse[b: Bool] { b in False }
9
10 fun Not[b: Bool] : Bool {
11   Bool - b
12 }
13
14 fun And[b1, b2: Bool] : Bool {
15   subset_[b1 + b2, True]
16 }
17
18 fun Or[b1, b2: Bool] : Bool {
19   subset_[True, b1 + b2]
20 }
21
22 fun Xor[b1, b2: Bool] : Bool {
23   subset_[Bool, b1 + b2]
24 }
25
26 fun Nand[b1, b2: Bool] : Bool {
27   subset_[False, b1 + b2]
28 }
29
30 fun Nor[b1, b2: Bool] : Bool {
31   subset_[b1 + b2, False]
32 }
33
34 fun subset_[s1, s2: set Bool] : Bool {
35   (s1 in s2) => True else False
36 }
```

Figure A.1: Boolean Alloy model

```
1  module fsm
2
3  sig Node{
4     node_name: String,
5     type: String
6  }
7  sig Transition{
8     trans_name: String,
9     startsAt: Node,
10    endsAt: Node
11 }
12
13 fact{
14    ("start" + "state" + "stop") in String
15 }
16
17 pred Node_Name_isUnique(){
18    all n1, n2: Node | n1 != n2 <=> n1.node_name != n2.node_name
19 }
20 pred NodeType_isValid(){
21    all n: Node | n.type in ("start" + "state" + "stop")
22 }
23 pred Transition_Name_isUnique(){
24    all t1, t2: Transition | t1 != t2 <=> t1.trans_name != t2.trans_name
25 }
26 pred startsAt_endsAt_isNodeID(){
27    Transition.startsAt in Node
28    Transition.endsAt in Node
29 }
30 pred StopNode_isNotStartsAt(){
31    all n: Node | n.type = "stop" => n not in Transition.startsAt
32 }
33 pred StartNode_isNotEndsAt(){
34    all n: Node | n.type = "start" => n not in Transition.endsAt
35 }
36 pred StartNode_StopNode_isUnique(){
37    one n: Node | n.type = "start"
38    one n: Node | n.type = "stop"
39 }
40 pred NodeisReachableFromStartNode(){
41    let next = {n1: Node, n2: Node | some t: Transition | t.startsAt = n1 && t.endsAt = n2}{
42       one n: Node {
43          n.type = "start"
44          Node = n.*next
45       }
46    }
47 }
48 pred StopNodeisReachableFromNode(){
49    let next = {n1: Node, n2: Node | some t: Transition | t.startsAt = n1 && t.endsAt = n2}{
50       all n1, n2: Node | n2.type = "stop" => n2 in n1.*next
51    }
52 }
53 pred repOk(){
54    Node_Name_isUnique
55    NodeType_isValid
56    Transition_Name_isUnique
57    startsAt_endsAt_isNodeID
58    StopNode_isNotStartsAt
59    StartNode_isNotEndsAt
60    StartNode_StopNode_isUnique
61    NodeisReachableFromStartNode
62    StopNodeisReachableFromNode
63 }
64
65 run repOk for 3 Node, 2 Transition
```

Figure A.2: Finite state machine Alloy model

```
module cd

abstract sig Object{
   name: String,
}
sig Class extends Object{}
sig Interface extends Object{}
sig Association{
   from: one Object,
   to: one Object,
   type: String
}

fact{
   ("depends_on" + "inherits_from" + "implements" + "is_associated_with"
   + "is_an_aggregate_of" + "is_composed_of") in String
}

pred object_name_isUnique(){
   all o1, o2: Object | o1 != o2 <=> o1.name != o2.name
}
pred no_multiple_inheritance(){
   all a1, a2: Association |
   (a1 != a2 && a1.type = "inherits_from" && a2.type = "inherits_from") =>
   ((a1.to != a2.to) => (a1.from != a2.from))
}
pred inheritsFrom_implements_isUnique(){
   all a1: Association | no a2: Association{
      a1.type in ("inherits_from" + "implements") && a1 != a2 &&
      a1.from = a2.from && a1.to = a2.to && a1.type = a2.type
   }
}
pred inheritsFrom_implements_isAcyclic(){
   let next = {o1: Object, o2: Object | some a: Association |
   a.from = o1 && a.to = o2 && (a.type = "inherits_from" || a.type = "implements")}
   { all o: Object | o !in o.^next }
}
pred implements_fromClass_toInterface(){
   all a: Association |
   a.type = "implements" => (some c: Class | some i: Interface | a.from = c && a.to = i)
}
pred inheritsFrom_fromtoClass_fromtoInterface(){
   all a: Association |
   a.type = "inherits_from" =>
   ((some c1, c2: Class | a.from = c1 && a.to = c2) ||
   (some i1, i2: Interface | a.from = i1 && a.to = i2))
}
pred association_fromClass_toClass(){
   all a: Association |
   (a.type !in ("inherits_from" + "implements")) =>
   (some c1, c2: Class | a.from = c1 && a.to = c2)
}
pred aggregation_composition_doNotConflict(){
   all a1, a2: Association |
   (a1.type = "is_an_aggregate_of" && a2.type = "is_composed_of") =>
   (a1.from != a2.from || a1.to != a2.to)
}
pred composition_isAcyclic(){
   let next = {o1: Object, o2: Object | some a: Association |
   a.from = o1 && a.to = o2 && a.type = "is_composed_of"}
   { all o: Object | o !in o.^next }
}
pred repOk(){
   object_name_isUnique
   no_multiple_inheritance
   inheritsFrom_implements_isUnique
   inheritsFrom_implements_isAcyclic
   implements_fromClass_toInterface
   inheritsFrom_fromtoClass_fromtoInterface
   association_fromClass_toClass
   aggregation_composition_doNotConflict
   composition_isAcyclic
}

run repOk for 2 Class, 1 Interface, 2 Association
```

Figure A.3: Class diagram Alloy model for Java

Bibliography

[1] Alloy Reference. http://alloy.mit.edu/alloy/documentation/book-chapters/alloy-language-reference.pdf.

[2] Software FDR2. http://www.fsel.com/software.html.

[3] Bernhard K. Aichernig, Percy Antonio, and Pari Salas. Test case generation by ocl mutation and constraint solving. In *In QSIC*, pages 64–71. IEEE Computer Society, 2005.

[4] Bernhard K. Aichernig and He Jifeng. Mutation testing in utp. *Form. Asp. Comput.*, 21(1-2):33–64, February 2009.

[5] Emina Torlak And. Kodkod for alloy users.

[6] Ellen Francine Barbosa, Jos Carlos Maldonado, and Auri Marcelo Rizzo Vincenzi. Toward the determination of sufficient mutant operators for c. *Software Testing, Verification and Reliability*, 11(2):113–136, 2001.

[7] Fevzi Belli and Oliver Jack. Declarative paradigm of test coverage. *Software Testing, Verification and Reliability*, 8(1):15–47, 1998.

[8] Vidroha Debroy and W. Eric Wong. Using mutation to automatically suggest fixes for faulty programs. *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, 0:65–74, 2010.

[9] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, April 1978.

[10] Gordon Fraser and Andreas Zeller. Mutation-driven generation of unit tests and oracles. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 147–158, New York, NY, USA, 2010. ACM.

[11] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *IEEE Trans. Software Eng.*, 38(1):54–72, 2012.

[12] R.G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, 3(4):279–290, 1977.

[13] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for 8 each. pages 3–13, June 2012.

[14] Yu-Seung Ma, Jeff Offutt, and Yong-Rae Kwon. Mujava: A mutation system for java. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 827–830, New York, NY, USA, 2006. ACM.

[15] A.P. Mathur. Performance, effectiveness, and reliability issues in software testing. In *Computer Software and Applications Conference, 1991.*

COMPSAC '91., Proceedings of the Fifteenth Annual International, pages 604–605, Sep 1991.

[16] Vajih Montaghami and Derek Rayside. Extending alloy with partial instances. In *Proceedings of the Third International Conference on Abstract State Machines, Alloy, B, VDM, and Z*, ABZ'12, pages 122–135, Berlin, Heidelberg, 2012. Springer-Verlag.

[17] A.S. Namin, J.H. Andrews, and D. Murdoch. Sufficient mutation operators for measuring test effectiveness. In *Software Engineering, 2008. ICSE '08. ACM/IEEE 30th International Conference on*, pages 351–360, May 2008.

[18] Tim Nelson, Salman Saghafi, Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. Aluminum: Principled scenario exploration through minimality. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 232–241, Piscataway, NJ, USA, 2013. IEEE Press.

[19] A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H. Untch, and Christian Zapf. An experimental determination of sufficient mutant operators. *ACM Trans. Softw. Eng. Methodol.*, 5(2):99–118, April 1996.

[20] A. Jefferson Offutt and Ronald H. Untch. Mutation testing for the new century. chapter Mutation 2000: Uniting the Orthogonal, pages 34–44. Kluwer Academic Publishers, Norwell, MA, USA, 2001.

[21] David Schuler and Andreas Zeller. Javalanche: Efficient mutation testing for java. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 297–298, New York, NY, USA, 2009. ACM.

[22] Ilya A Shlyakhter. Declarative symbolic pure-logic model checking. Technical report, 2005.

[23] T. Srivatanakul, John A. Clark, S. Stepney, and F. Polack. Challenging formal specifications by mutation: a csp security example. In *Software Engineering Conference, 2003. Tenth Asia-Pacific*, pages 340–350, Dec 2003.

[24] Allison Sullivan, Razieh Nokhbeh Zaeem, Sarfraz Khurshid, and Darko Marinov. Towards a test automation framework for alloy. In *Proceedings of the 2014 International SPIN Symposium on Model Checking of Software*, SPIN 2014, pages 113–116, New York, NY, USA, 2014. ACM.

[25] Lingming Zhang, Darko Marinov, Lu Zhang, and Sarfraz Khurshid. Regression mutation testing. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 331–341, New York, NY, USA, 2012. ACM.